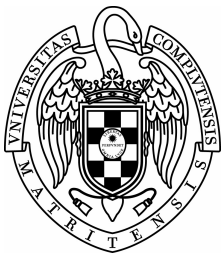


Sistemas Informáticos
Curso 2004-2005

Implementación de un Algoritmo Genético Paralelo sobre HW Gráfico de última generación

Carmen Córdoba González
Juan Carlos Pedraz Sánchez

Dirigido por:
Prof. José Ignacio Hidalgo Pérez
Prof. Christian Tenllado Van der Reijden
Dpto. Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

Asignatura de Sistemas Informáticos. Curso 2004/2005

Proyecto: Implementación de un AG paralelo sobre hardware gráfico de última generación

Dirigido por los profesores D. José Ignacio Hidalgo Pérez y D. Christian Tenllado

Carmen Córdoba González

Juan Carlos Pedraz Sánchez

Madrid, 30 de Septiembre de 2005.

Palabras Clave

AG

Algoritmo

Genético

GPU

Gráfica

Paralelización

Paralelo

Rendimiento

Tarjeta

Resumen

Español

Los algoritmos genéticos (AGs) son procedimientos de búsqueda y optimización inspirados por la simplicidad y efectividad del proceso de la evolución natural de las especies. Al igual que ocurre en la naturaleza, basan su éxito en la supervivencia de los individuos más aptos de una población. En este caso, un individuo es una solución potencial del problema, y se implementa como una estructura de datos. Los AGs trabajan sobre poblaciones de soluciones que evolucionan mediante la aplicación de operadores genéticos (selección, cruce y mutación) adecuados al problema específico que intentan resolver. Uno de los rasgos esenciales de los AGs es su paralelismo implícito puesto que, al igual que la evolución natural, trabajan con poblaciones enteras, no con sus individuos integrantes en particular.

Los sistemas actuales cuentan con potentes tarjetas gráficas, frecuentemente programables, que permanecen inactivas durante la ejecución de aplicaciones no gráficas. Dichas tarjetas cuentan con un procesador de naturaleza paralela, que las hace especialmente indicadas para la ejecución de AGs.

Este trabajo es una propuesta para aprovechar un recurso hardware habitualmente inactivo para implementar, en un sistema monoprocesador, algún tipo de topología de AGs paralelos. Obtenemos así mejoras -tanto en la calidad de las soluciones como en el tiempo de ejecución- con respecto a la ejecución secuencial del AG sobre la CPU.

English

Genetic algorithms (GAs) are optimization techniques which imitate the way that nature selects the best individuals (the best adaptation to the environment) to create descendants which are more highly adapted. The first step is to generate a random initial population, where each individual is represented by a character chain like a chromosome and with the greatest diversity, so that this population has the widest range of characteristics. Each individual represents a solution for the targeted problem. Then, each individual is evaluated using a fitness function, which indicates the quality of each individual. Finally, the best-adapted individuals are selected to generate a new population, whose average will be nearer to the desired solution. This new population is created making use of three operators: selection, crossover and mutation.One of the major aspects of GA is their ability to be parallelised. Indeed, because natural evolution deals with an entire population and not only with particular individuals, it is a remarkably highly parallel process.

Nowadays computer systems incorporate powerful graphic cards that are commonly idle during a normal execution process of most of the optimization algorithms. Modern graphic cards use a pipelined streaming architecture to perform a significant part of the rendering process. Two stages in the pipelined process are programmable in current graphics hardware. The vertex engine is used to perform transformations on the vertex attributes (normal, position, color, texture, ...). On the other hand, the fragment engine is used to transform the fragments that form the different polygons. Both engines are extremely parallel, processing several elements in parallel and making extensive use of SIMD units.

In this work we have presented a parallel implementation of a GA using a GPU. We have implemented not only three well know benchmarks problems with excellent Speed-up results, but also a novel implementation of an algorithm for solving defectives problems proposed in the literature.

Agradecimientos

Gracias a José Ignacio Hidalgo Pérez y Christian Tenllado por su apoyo, tutelaje y total disponibilidad en la realización de este trabajo. Y por su paciencia ante los continuos abordajes en el momento y lugar menos esperados.
Agradecemos también el apoyo técnico y logístico prestado por Carlos Roa.

Este trabajo ha sido financiado parcialmente por el proyecto del Ministerio de Ciencia y Tecnología TIC 2002-750 del Gobierno de España.

Índice

1 Introducción

1.1 Objetivos

Los algoritmos genéticos se plantearon como una prometedora técnica genérica de resolución de problemas de optimización. Inspirados por la simplicidad y efectividad del proceso de la evolución natural de las especies, demostraron contar con un enorme potencial y definieron una interesante vía de investigación. Una vez quedó consolidada la validez y las posibilidades resolutorias de este tipo de algoritmos, comenzó a plantearse el problema de la optimización de los mismos y a sugerirse posibles formas de reducir los tiempos de ejecución. Existe una basta bibliografía en este sentido, donde se hacen diversas propuestas para minimizar los recursos utilizados por el AG, tanto en memoria de almacenamiento como en tiempo de ejecución y de convergencia de soluciones.

Uno de los rasgos esenciales de los AGs es su paralelismo implícito puesto que, al igual que la evolución natural, trabajan con poblaciones enteras, no con sus individuos integrantes en particular (Goldberg 1989a). Esta característica ha dirigido la mayoría de propuestas, para la mejora del rendimiento, hacia la ejecución en computadores paralelos o en sistemas que los emulen (por ejemplo computadores conectados vía Internet).

Sin embargo, la tecnología paralela está aún fuera del alcance de muchos usuarios. Por otro lado, también se ha investigado mucho en este sentido, y no es fácil aportar nuevas ideas en este campo.

Nuestro trabajo se enmarca en sistemas computacionales más extendidos y accesibles para usuarios que trabajan habitualmente con AGs sobre sistemas monoprocesador. Los sistemas actuales cuentan con potentes tarjetas gráficas, frecuentemente programables, que permanecen inactivas durante la ejecución de aplicaciones no gráficas. Dichas tarjetas cuentan con un procesador (habitualmente dos) de naturaleza paralela (los programas se ejecutan sobre cada elemento de datos) que en principio parece muy apropiado para la ejecución de AGs. Por tanto, podemos aprovechar un recurso hardware habitualmente inactivo para implementar en un sistema monoprocesador algún tipo de topología de AGs paralelos, llevando múltiples poblaciones interconectadas, y obteniendo así mejoras -tanto en la calidad de las soluciones como en el tiempo de ejecución- con respecto a una ejecución tradicional secuencial del AG sobre la CPU.

El objetivo que nos planteamos al comenzar el desarrollo del presente trabajo es demostrar que la GPU puede ser un sistema eficiente para procesar AGs,

implementando un AG paralelo con dos poblaciones que se comuniquen entre sí. Una población se procesará en la CPU y la otra en la GPU. La planificación y control general del proceso evolutivo deberá realizarse desde la CPU, dadas las características de la GPU, pero teniendo en cuenta que la mayor parte del esfuerzo computacional que se realiza en un AG se debe a la operación de evaluación de los individuos (Povinelli & Feng 1999b), la mejora promete ser considerable.

1.2 Motivación

Los algoritmos genéticos han demostrado ser una efectiva herramienta de resolución para afrontar complejos problemas de optimización y de aprendizaje máquina [85]. Los AGs demuestran capacidad de búsqueda global, gracias a que evalúan simultáneamente aptitudes en múltiples puntos del espacio de soluciones.

Al igual que el proceso evolutivo natural, el algoritmo genético trabaja sobre una población de individuos codificados, que representan posibles soluciones aleatorias al problema abordado. A cada individuo se le asigna una función de aptitud que indica la calidad de la solución candidata (en términos del problema que se desea solucionar), al igual que en la naturaleza se puede hablar de individuos más aptos (o adaptados) para un determinado entorno o hábitat.

En cada generación del AG se evalúan múltiples candidatos, y la búsqueda es dirigida de acuerdo al principio natural de la *supervivencia del más apto*. Para producir la siguiente generación de soluciones candidatas se procede a intercambiar (reproducción de los individuos) y modificar (mutación genética) información de búsqueda y coordenadas. El ciclo evolutivo se repite hasta que se alcanza la solución al problema o algún límite fijado previamente.

El esfuerzo computacional involucrado en un proceso evolutivo como el descrito es enorme y tiene además un inherente carácter de búsqueda paralela. En particular, para problemas complejos con una población lo suficientemente grande para la que se realizan muchas generaciones, la cantidad de trabajo computacional necesaria para simular un modelo evolutivo realista y encontrar las soluciones óptimas globales es enorme [85].

Se han realizado algunas implementaciones relacionadas con explotar el paralelismo implícito de los AGs creando infraestructuras que soportan el procesamiento de información distribuida, usando por ejemplo los recursos hardware disponibles e Internet (dividiendo una tarea en subtareas y resolviendo éstas simultáneamente usando varios procesadores sobre Internet). Muchas de las implementaciones de AG paralelos (AGP) han buscado hacer un uso óptimo de arquitecturas paralelas, por ejemplo Talbi et al presentaron un AGP para la resolución del problema de la partición de grafos. Para ello utilizaron una implementación sobre un Supernodo de Transputers [92]. El Supernodo consistía en una máquina altamente paralela basada en microprocesadores de 32-bit con una memoria on-chip y una unidad de Punto Flotante. La comunicación entre transputers se realizaba mediante cuatro conexiones serie asíncronas bidireccionales de punto a punto.

Hidalgo et al. utilizan un AGP implementado sobre una supercomputadora Cray T3E para resolver un problema de diseño de circuitos sobre FPGAs [93]. Existen numerosas implementaciones de AGPs sobre clusters de computadoras con resultados excelentes.

Recientemente Herrera et al. han presentado una implementación orientada a entornos Grid. La mayoría de las dificultades de implementación surgen de las características propias del Grid como son la complejidad, heterogeneidad, dinamismo y alto porcentaje de fallos. Esto hace que no sea una implementación muy apropiada para problemas de optimización que requieran una solución rápida [94].

En lugar de evolucionar la población completa sobre un solo procesador, se introdujo el concepto de varias subpoblaciones intercomunicadas, en analogía con la evolución natural de poblaciones distribuidas espacialmente. Esta comunicación permite que los individuos migren entre las subpoblaciones basándose en ciertos patrones, introduciendo así diversidad de individuos de élite (aquellos con mayor aptitud) periódicamente, de un modo que simula la evolución de las especies en el entorno natural.

Nos encontramos en medio de una gran transición en gráficos por computador, tanto en términos del hardware gráfico como en términos de la calidad visual y elaboración de juegos, aplicaciones interactivas, y animaciones. El hardware gráfico ha evolucionado desde grandes y pesadas estaciones de trabajo gráficas metálicas que costaban cientos de miles de dólares hasta unidades de procesamiento gráfico (GPUs) de chips simples cuyo rendimiento y características han crecido hasta alcanzar e incluso superar los de las estaciones de trabajo tradicionales.

La potencia de procesamiento provista por una GPU moderna en un marco simple rivaliza la cantidad de computación que suele gastarse para un marco de animación renderizado offline. De hecho, en el lanzamiento de GeForce3 en el Apple de Macintosh, una convincente versión de Luxo Jr. (cortometraje de animación de Pixar) demostró ejecutarse interactivamente en tiempo real. En la conferencia SIGGRAPH 2001 una versión interactiva de una película más reciente, Final Fantasy (de Square Studios), se reprodujo en tiempo real de nuevo en una GeForce3.

Además de la gran potencia de cálculo de la tarjeta gráfica en términos absolutos, las GPUs cuentan con una característica que las hace especialmente indicadas para ejecutar AGs: los programas se ejecutan paralelamente sobre cada elemento de datos definido.

Toda la potencia de cálculo que nos ofrecen las GPUs actuales, así como la idoneidad de su carácter paralelo, permanece desaprovechada cuando ejecutamos un AG de la forma habitual sobre la CPU. El uso de todos los procesadores programables con que cuenta nuestro sistema promete mejorar el rendimiento, y especialmente cuando uno de los procesadores disponibles cuenta con unas características tan adecuadas para el tipo de algoritmo a ejecutar.

Así pues, trabajando con un sistema monoprocesador al que cualquier tipo de usuario puede tener acceso, lograremos las ventajas de la ejecución de AGs en sistemas multiprocesador, manteniendo dos subpoblaciones intercomunicadas que evolucionan

en paralelo. Presumiblemente la ejecución sobre la GPU irá más rápida y, por tanto, en un mismo intervalo de tiempo realizará más generaciones que la CPU. Es de esperar que en este caso la GPU obtenga mejores soluciones que la CPU pero, gracias a la migración periódica (en la que se intercambian los mejores individuos), estos resultados de mayor calidad serán compartidos por ambos procesadores y afectarán positivamente al proceso evolutivo completo.

En este trabajo se han realizado implementaciones de AGs paralelos para resolver los siguientes problemas:

- onemax
- función multimodal
- función defectiva

Además se ha implementado un AG modificado que soluciona el problema defectivo.

Los prototipos desarrollados se han sometido a grandes baterías de pruebas, obteniéndose resultados prometedores.

El resto de la memoria está organizado como sigue:

- En el capítulo 2 se da un repaso de los algoritmos genéticos simples y paralelos, y se describe el método implementado para solucionar el problema defectivo (algoritmo de mapeo evolutivo).
- En el capítulo 3 se describe el tipo de hardware gráfico utilizado, así como las características que han sido determinantes en el proceso de desarrollo de los prototipos.
- En el capítulo 4 se describe el proceso de implementación y desarrollo de los algoritmos, tanto sobre la CPU como sobre la GPU.
- En el capítulo 5 se presentan los resultados experimentales obtenidos mediante baterías de pruebas de las implementaciones realizadas, así como las conclusiones que sugieren.
- Por último, el capítulo 6 cierra la presente memoria con un resumen de las conclusiones que se desprenden del trabajo realizado, una revisión de las tesis y objetivos iniciales y el planteamiento de futuros trabajos.
- Los apéndices incluidos recopilan detalles que pueden resultar útiles para desarrollos de proyectos similares, o para ampliaciones de éste.

2 Algoritmos Genéticos (AGs)

Durante los últimos treinta años, ha habido un creciente interés en los sistemas de resolución de problemas basados en los principios de la evolución y la herencia. Dichos sistemas mantienen una población de soluciones potenciales, cuentan con algún proceso de selección basado en la aptitud (fitness) de los individuos, y aplican un conjunto de operadores *genéticos* [21].

Desde que se introdujeron estos conceptos, se han desarrollado numerosos tipos de estos sistemas que usan estrategias evolutivas para resolver problemas de optimización parametrizados. Los primeros sistemas basados en evolución que se plantearon fueron las estrategias evolutivas, que trabajan con un número muy reducido de individuos. Le siguieron los algoritmos genéticos de Holland[25], en cuya descripción profundizaremos en los siguientes apartados. La programación evolutiva de Fogel [23] es una técnica de búsqueda a través de un espacio de pequeños autómatas de estados finitos. Las técnicas de búsqueda dispersa de Glover [24] mantienen una población de puntos de referencia y generan su descendencia mediante combinaciones lineales con pesos. En 1990, Koza propuso unos sistemas basados en la evolución (Programación Genética) para buscar el programa de computadora más apto para resolver un problema particular. Se usa el término común Programas Evolutivos (PEs) para todos los sistemas basados en evolución.

Programas evolutivos

Un programa evolutivo es un algoritmo probabilístico que mantiene una población de individuos, $P(t) = \{x_1^t, ..., x_n^t\}$ para la iteración t. Cada individuo representa una solución potencial al problema, y está implementado como alguna estructura de datos S (posiblemente compleja). La población inicial puede generarse aleatoriamente o crearse mediante algún proceso heurístico, y es el punto de partida para el programa evolutivo. Cada solución x_i^t es evaluada para dar alguna medida de su aptitud (fitness). Después, se forma una nueva población (iteración t+1) seleccionando los individuos más aptos (paso de selección). Algunos miembros de la nueva población sufren transformaciones (paso de cambio), debido a operadores genéticos, para formar nuevas soluciones. Hay transformaciones unarias m_i (mutación), que crean nuevos individuos a partir de un pequeño cambio en un individuo simple ($m_i : S \to S$), y transformaciones de orden superior c_j (cruce), que producen nuevos individuos combinando partes de varios (dos o más) individuos ($c_j : S \times ... \times S \to S$). Tras varias generaciones, el programa converge, y se espera que el mejor individuo represente una solución razonablemente próxima al óptimo [21].

La función de evaluación incorpora los requerimientos del problema, y devuelve la aptitud de cada individuo, distinguiéndose así entre individuos mejores y peores.

Los operadores genéticos pueden diseñarse de diferentes formas. Uno de los problemas que pueden aparecer tras la aplicación de los operadores es que aparezcan nuevos individuos que no satisfagan las restricciones del problema, o que no sean representaciones válidas de soluciones. Este contratiempo puede solucionarse incorporando a los operadores conocimiento específico del problema a resolver. La otra posibilidad es designar operadores independientes del problema, e incorporar los requerimientos a la función de evaluación, penalizando así los individuos que no se ajusten a lo deseable [21].

Evidentemente, para un problema determinado pueden formularse diferentes programas evolutivos. Dichos programas pueden diferir en distintos aspectos: pueden usar diferentes estructuras de datos para implementar individuos, diferentes operadores genéticos para transformar los individuos, diferentes métodos para crear la población inicial, diferentes formas de manejar las restricciones del problema y diferentes parámetros (tamaño de población, probabilidad de aplicar los diferentes operadores, número de generaciones, etc.). Sin embargo, todos ellos comparten un principio común: una población de individuos sufre algunas transformaciones, y durante este proceso evolutivo estos individuos luchan por sobrevivir.

Algoritmos genéticos

Los algoritmos genéticos (AGs) son procedimientos de búsqueda y optimización inspirados en el concepto de evolución que se observa en el mundo biológico. Al igual que ocurre en la naturaleza, basan su éxito en la supervivencia de los individuos más aptos de una población. En este caso, un individuo es una solución potencial del problema y se implementa como una estructura de datos. Los AGs trabajan sobre poblaciones de soluciones que evolucionan de generación en generación gracias a una serie de operadores genéticos (selección, cruce y mutación) adecuados al problema específico que intentan resolver.

Los algoritmos genéticos fueron introducidos por John Holland en los años 60, y desarrollados posteriormente por su grupo de investigación en la Universidad de Michigan [49]. El AG de Holland es un método para pasar de una población (conjunto) de individuos (soluciones) representados por cromosomas (cadenas de bits que representan soluciones a un problema) a una nueva población. El mecanismo usado para progresar a nuevas poblaciones emula la selección natural, y también se introducen sendos operadores inspirados en el cruce o reproducción entre los individuos seleccionados (operador cruce) y la mutación aleatoria de los genes de los individuos resultantes del cruce (operador mutación).

Cada individuo está representado por un cromosoma, que es una cadena de genes donde cada gen tiene un valor concreto de entre los posibles valores de dicho gen, llamados alelos. La codificación se realiza en cadenas de bits, de modo que cada gen puede tener un determinado alelo de entre 0 y 1.

La selección se encarga de escoger los individuos que participarán en la formación de la nueva población, basándose principalmente en la aptitud de estos individuos para resolver el problema.

El operador de cruce combina dos individuos seleccionados, formando uno nuevo que comparte genes de ambos y que, por tanto, tiene en su cromosoma parte de los cromosomas de sus dos progenitores.

El operador de mutación cambia azarosamente el valor de algunos genes de los individuos que van a pasar a formar parte de la siguiente población. De este modo, logra introducir cambios aleatorios en la población y ampliar así el espacio de búsqueda.

Por último, Holland introdujo también el operador de inversión, que invierte el orden de una parte del cromosoma, y que nosotros repasaremos brevemente, puesto que normalmente no se incorpora en los AGs actuales.

Programas evolutivos VS Algoritmos genéticos

Los algoritmos genéticos son un tipo de programa evolutivo. La estructura de un algoritmo genético es la misma que la de un programa evolutivo, y las diferencias entre ambos tipos están ocultas en niveles inferiores.

En los PEs no es necesario representar los cromosomas mediante cadenas de bits, y el proceso de cambios incluye otros operadores genéticos apropiados para la estructura y el problema dados. De Jong dice en [26] que cuando los elementos del espacio a buscar son representados de forma más natural por estructuras de datos más complejas que listas, árboles, grafos, etc., en vez de intentar linealizarlos en una representación de cadena debiera intentarse redefinir el cruce y la mutación para trabajar directamente en dichas estructuras complejas. Sin embargo, esta nueva orientación no parece *cuajar* definitivamente.

Por el contrario, los AGs usan cadenas binarias de longitud fija y tienen dos operadores genéticos básicos.

En principio parece que una representación natural de las soluciones potenciales a un problema dado, junto con una familia de operadores genéticos aplicables debiera ser indiscutiblemente útil en la resolución de muchos problemas, y que esta aproximación es una dirección prometedora de resolver problemas de modo genérico. Además de otros paradigmas de computación evolutiva (estrategias evolutivas, programación evolutiva, programación genética, etc.), algunos investigadores en algoritmos genéticos han explorado el uso de otras representaciones como listas ordenadas, listas embebidas o listas de elementos variables. En los últimos años se han descrito diferentes variaciones de aplicación específica en AGs [27, 28, 29, 30, 31, 32, 33, 34]. Estas variaciones incluyen cadenas de longitud variable, estructuras más ricas que las tradicionales cadenas binarias (por ejemplo matrices [34]), y experimentos con operadores genéticos modificados. Así pues, parece que la mayoría de los investigadores han modificado sus implementaciones de algoritmos genéticos bien usando una representación de cromosomas diferente, bien diseñando operadores genéticos específicos para acomodarse al problema específico. Koza dice en [35] que la

representación es clave en los AGs debido a que el esquema de representación puede limitar severamente la *ventana a través de la que el sistema observa el mundo*. Los esquemas de representación basados en cadenas resultan difíciles y poco naturales para muchos problemas, y en diversas ocasiones ([26, 36, 37]) se ha reconocido la necesidad de representaciones más potentes. Sin embargo, en [38] se recuerda que en el trabajo de Holland (y en el de muchos de sus estudiantes) los cromosomas son cadenas de bits.

Michalewicz se pregunta en [21] el porqué de esta tendencia, partiendo desde los AGs, hacia programas evolutivos más flexibles. Su respuesta es que, aunque cuenten con una buena base teórica, los AGs han fallado en proporcionar aplicaciones exitosas en muchas áreas. Parece ser que el principal factor causante del fracaso de los AGs es también el responsable de su éxito: la independencia del dominio.

Una de las consecuencias de la pulcritud de los AGs, en cuanto a esta independencia del dominio, es su escasa habilidad para tratar restricciones no triviales. Como dijimos anteriormente, una de las cuestiones clave a la hora de diseñar una representación en cromosomas de las soluciones a un problema es la implementación de las restricciones en las soluciones (conocimiento específico del problema). Las restricciones que no pueden violarse pueden implementarse de tres formas, según [38], aunque cada una tiene sus desventajas:

1) Imponiendo grandes penalizaciones a los individuos que las violen.

   Se corre el riesgo de crear un AG que gasta la mayor parte de su tiempo evaluando individuos *ilegales*. Lo que es más, puede ocurrir que cuando se encuentra un individuo legal, la población converja a él sin buscar individuos mejores. Esto se debe a que los caminos que llevan a individuos legales mejores pueden requerir la producción de individuos ilegales como pasos intermedios, y las penalizaciones que se les imponen a éstos impiden que se reproduzcan.

2) Imponiendo penalizaciones moderadas.

   El sistema puede desarrollar individuos que violen las restricciones pero que son mejor valorados que otros individuos legales. Esto se debe a que el resto de la función de evaluación puede ser satisfecha mejor aceptando la penalización moderada por restricciones que evitando la penalización.

3) Creando decodificadores que eviten individuos ilegales.

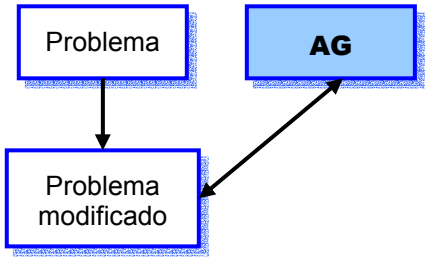
   La construcción, en el procedimiento de evaluación, de decodificadores que impidan la creación de individuos ilegales a partir de sus cromosomas suele resultar en un algoritmo con demasiada intensidad de cómputo. Además, no todas las restricciones se pueden implementar fácilmente de este modo.

En [21] pueden encontrarse ejemplos de decodificadores y algoritmos de reparación, así como diversas funciones de penalización.

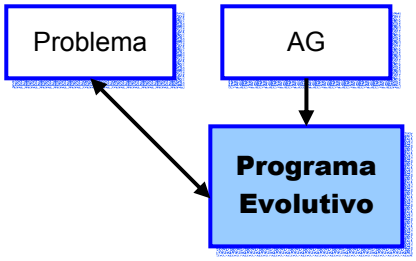
En los primeros pasos de la inteligencia artifical (IA), se plantearon los *resolvedores* generales de problemas (General Problem Solvers, GPSs) como herramientas genéricas para la solución de problemas complejos. Sin embargo, fue viéndose que era necesario incorporar conocimiento específico del problema debido a la complejidad inmanejable

de estos sistemas. Algo similar ha ocurrido con los AGs, y numerosos investigadores han reconocido la necesidad de incorporar conocimiento específico [39, 40, 41, 42, 43]. Parece que los AGs (vistos como GPSs) son demasiado independientes del dominio como para resultar útiles en muchas aplicaciones. Por esto los programas evolutivos que incorporan conocimiento específico en las estructuras de datos de los cromosomas y en los operadores genéticos parecen funcionar mejor [21].

La diferencia conceptual básica entre algoritmos genéticos clásicos y los programas evolutivos se refleja en los siguientes esquemas:



Aproximación del Algoritmo Genético



Aproximación del Programa Evolutivo

Los AGs clásicos operan con cadenas binarias y precisan de una traducción del problema original a la forma apropiada. Esta tarea puede incluir mapeo entre las soluciones potenciales y su representación binaria, uso de decodificadores o algoritmos de reparación, etc. En general no es labor fácil.

Los programas evolutivos dejarían el problema sin modificar, modificando la representación cromosómica de las soluciones potenciales (usando estructuras de datos naturales) y aplicando operadores genéticos apropiados.

De lo anterior se desprende que, para resolver un problema usando un programa evolutivo, podríamos bien transformar el problema a una forma adecuada para los AGs (figura de la izquierda), bien transformar el AG para ajustarlo al problema (figura de la derecha)

Es muy difícil trazar una línea divisoria entre AGs y PEs, y muchos investigadores han encontrado un gran potencial tras varias modificaciones. En [44] Davis *hibrida* un algoritmo existente con la técnica de AG, y define los *algoritmos genéticos híbridos* como aquellos algoritmos que aplican los siguientes tres principios:

- Usan la codificación actual: usan la técnica de codificación del algoritmo original en el algoritmo híbrido.
- Hibridan cuando es posible: incorporan las características positivas del algoritmo original en el algoritmo híbrido.
- Adaptan los operadores genéticos: se crean los operadores de cruce y mutación para el nuevo tipo de codificación mediante analogía con los operadores de cruce y mutación de las cadenas de bits. También incorporan heurísticas basadas en el dominio a modo de operadores.

A la vista de las características descritas, los AGs híbridos parecen compartir muchos conceptos con los programas evolutivos.

2.1 Algoritmos genéticos Simples

Hay una gran clase de problemas interesantes para los que aún no se han desarrollado algoritmos razonablemente rápidos. Muchos de estos problemas son problemas de optimización que surgen frecuentemente en diversas aplicaciones.

En general, cualquier tarea abstracta a completar puede tomarse como la resolución de un problema, que a su vez puede considerarse una búsqueda en un espacio de soluciones potenciales. Puesto que buscamos la mejor solución, podemos ver nuestra tarea como un proceso de optimización [21].

Para los espacios de búsqueda pequeños, los métodos exhaustivos tradicionales suelen bastar; sin embargo, para espacios mayores se necesitan técnicas de inteligencia artificial especiales. Los algoritmos genéticos (AGs) pertenecen a este último tipo de técnicas, y son algoritmos estocásticos cuyos métodos de búsqueda imitan fenómenos naturales: la herencia genética y la lucha Darwiniana por la supervivencia.

La metáfora que subyace a los AGs es la de la evolución natural. En la evolución, el problema que cada especie afronta es el buscar adaptaciones beneficiosas a un entorno complejo y cambiante. El conocimiento que cada especie ha conseguido a través del proceso es incorporado en la caracterización de los cromosomas de sus miembros [38].

Vocabulario de los AGs

Puesto que los AGs están inspirados en los mecanismos de evolución natural, es lógico que el vocabulario que manejan haya sido también importado de este campo de la ciencia. Así, se habla de individuos (o genotipos o estructuras) de una población. También es frecuente referirse a los individuos como cadenas o cromosomas.

! *Esta equivalencia no es correcta en la teoría de la genética, puesto que cada célula de cualquier organismo contiene cierto número de cromosomas. Podemos usar indistintamente cualquiera de los términos anteriores sólo si hablamos - como en la presente memoria - de individuos de un cromosoma, es decir de cromosomas haploides (frente a diploides).*

A su vez, los cromosomas están constituidos por unidades más sencillas, los genes, distribuidas en sucesión lineal. Cada gen controla la herencia de uno o más caracteres, y los genes de ciertos rasgos están localizados en determinadas posiciones del cromosoma (loci). Cada carácter de los individuos puede manifestarse de forma diferente, adquirir diferentes valores (por ejemplo, el color del pelo o de los ojos de un sujeto): Se dice que el gen puede estar en diferentes estados, llamados alelos.

Cada genotipo representa una solución potencial al problema, mientras que el significado de un cromosoma particular - el *fenotipo* - lo define externamente el usuario. El proceso de evolución corre sobre una población de cromosomas, correspondiéndose con una búsqueda a través de un espacio de soluciones potenciales. Esta búsqueda precisa de un equilibro entre dos objetivos aparentemente contradictorios: explotar las mejores soluciones y explorar el espacio de búsqueda [45].

La técnica de escalada (hillclimbing) es un ejemplo de estrategia que explota la mejor solución tras posibles mejoras. Por el contrario, descuida la exploración del espacio de búsqueda. En esencia, este algoritmo es aplica a un solo punto del espacio (punto actual), de modo que en cada iteración se elige un punto de la vecindad: si el nuevo punto obtiene una mejor puntuación, el punto actual pasa a ser este nuevo punto; en otro caso se selecciona y coteja otro vecino. El proceso finaliza cuando no puede mejorarse la aptitud obtenida para el punto actual. Salta a la vista que esta técnica sólo consigue óptimos locales, y que la bondad del óptimo alcanzado depende del punto de partida elegido. Una forma sencilla de paliar este problema, aunque de modo muy limitado, es ejecutar el algoritmo de escalada para muchos puntos de partida diferentes.

En el extremo contrario, las búsquedas aleatorias son ejemplos típicos de estrategias que exploran el espacio de búsqueda ignorando la explotación de las regiones prometedoras.

Los AGs son una clase de métodos de búsqueda de propósito general (independientes del dominio) que logran un notable equilibrio entre la exploración y la explotación del espacio de búsqueda. Pertenecen a la clase de los algoritmos probabilísticos, aunque son muy diferentes de los algoritmos aleatorios puesto que combinan elementos de la búsqueda dirigida y estocástica. Gracias a esta mezcla, los AGs son incluso más robustos que los métodos de búsqueda dirigidos existentes. Otra propiedad importante de los métodos basados en genética es que mantienen una población de soluciones potenciales, en contraposición al resto de métodos que procesan un punto simple en el espacio de búsqueda (como hemos visto, por ejemplo, en la escalada).

Los algoritmos genéticos han sido aplicados con gran éxito a problemas de optimización tales como planificación, control adaptativo, juegos, modelado cognitivo, problemas de transporte, problema del viajero, optimizaciones de consultas a BBDD, rutado de cables, etc. [21].

Sin embargo, De Jong advierte en [26] que, debido a que históricamente los AGs se han centrado en aplicaciones de optimización de funciones, es fácil caer en la trampa de percibir los AGs en sí mismos como herramientas de optimización. Esta suposición errónea puede llevarnos a la sorpresa y la decepción cuando fallan al buscar un óptimo aparentemente obvio en un espacio de búsqueda concreto. La sugerencia que este autor hace para evitar esta trampa perceptiva es pensar en los AGs como una simulación de un proceso natural (la evolución), y tener siempre en cuenta que como tal incorporan los objetivos y propósitos de este proceso natural. Con este enfoque en mente, y pensando en el proceso natural que sirve de inspiración a los AGs, resulta más fácil abandonar esta idea de optimizador de funciones.

Durante la última década, el significado de *optimización* se ha extendido más allá: los computadores actuales sólo pueden resolver de modo aproximado muchos problemas de optimización combinatorios de gran escala, o problemas de ingeniería con fuertes restricciones. Los AGs aspiran a resolver problemas tan complejos como estos.

Qué son los AGs

Como hemos vistos, los algoritmos genéticos simples se basan en un principio de la evolución: los mejores individuos tienen mayor probabilidad de reproducirse y sobrevivir que otros individuos menos adaptados al entorno [44, 51, 21]. Para emular este principio de la naturaleza, los AGs mantienen una población, o conjunto de soluciones, que evoluciona a través del tiempo, y que finalmente converge a una solución única. Estos individuos de la población se representan mediante un cromosoma binario que codifica las variables del problema que se desea resolver. En cada generación las soluciones relativamente *buenas* se reproducen, mientras que las soluciones relativamente *malas* mueren.

Normalmente se emplea un cromosoma simple, constituido por una cadena de bits de longitud fija. Hay variantes en los que se representa una solución empleando cromosomas de longitud variable, o incluso varios cromosomas. En cualquiera de estos modelos, se asocia a cada individuo un valor de una función de coste (valor de fitness) que mide la calidad de la solución, y que es precisamente lo que permite simular el concepto de individuos mejor adaptados y por consiguiente la selección. Se otorgará mayor probabilidad de ser seleccionados para la próxima generación, pasando sus propiedades a los nuevos individuos, a los individuos con mejor valor de la función de coste.

Así, los elementos fundamentales que hay que definir en un algoritmo genético ([21], [22]) son :

- Una función de coste que evalúe la aptitud o adaptación de los individuos, y que juega el papel del entorno.
- Una codificación (representación genética) para representar las soluciones al problema.
- Una forma de crear una población inicial de soluciones potenciales.
- Operadores genéticos que alteren la composición de los hijos: operador de selección, de cruce y de mutación.
- Los valores de varios parámetros que usan los algoritmos genéticos: las probabilidades con las que se aplican cada uno de los operadores, el tamaño de la población, el número máximo de generaciones, etc.

La estructura general de un AG simple es la misma que la de cualquier programa evolutivo:

```
generación de la población inicial
while no se cumpla la condición de parada do
   evaluación de los individuos
   selección
cruce
   mutación
end while
```

En primer lugar se genera una población inicial a partir de la que trabajar. Suele inicializarse de forma aleatoria, aunque puede emplearse cualquier otro método que pueda resultar más eficiente en el problema concreto a solucionar. Una vez obtenido el conjunto de individuos iniciales, se pasa al proceso fundamental del algoritmo (cuerpo de bucle), que consiste en evaluar la población actual (según la función de coste elegida) y seleccionar los individuos que intervendrán en la generación de la próxima generación. A continuación se aplican los operadores de cruce y mutación, obteniendo así la nueva población, sobre la que se repiten los mismos pasos. Este proceso se repite tantas veces como indique la condición de parada, que puede ser un número máximo de generaciones, la convergencia de la población, etc.

El cruce combina las características de dos cromosomas padres para formar dos hijos similares mediante el intercambio de segmentos de los padres. La intuición tras la aplicación del operador de cruce es el intercambio de información entre diferentes soluciones potenciales. La mutación altera arbitrariamente uno o más genes de un cromosoma seleccionado, modificando aleatoriamente con una probabilidad fijada. La intuición tras el operador de mutación es la introducción de variación extra en la población.

2.1.1 Cómo funcionan los AGs

En este apartado describiremos las acciones de un algoritmo genético sencillo, a fin de dar una idea general del proceso. En los sucesivos apartados se detallarán los diferentes elementos que intervienen en el algoritmo (operadores, codificación, etc.), así como las principales elecciones que podemos hacer. Como se irá viendo, los AGs permiten muchas posibilidades de implementación, y la elección de unas u otras dependerá del problema concreto a afrontar, de los recursos disponibles, o simplemente de las preferencias personales del desarrollador.

A fin de simplificar la explicación, haremos algunas suposiciones (por supuesto lícitas) sobre los problemas a resolver por un AG cualquiera [21]:

En primer lugar, podemos suponer sin ninguna pérdida de generalidad que abordaremos sólo problemas de maximización. Si se pretendiera solucionar un problema de minimización de una función f, esto sería equivalente a maximizar la función g = -f. Es decir: min f(x) = max g(x) = max {-f(x)}.

Además, asumiremos que las funciones objetivo f toman valores positivos en sus dominios. En otro caso bastaría con añadir una constante positiva C: max g(x) = max {f(x) + C}.

Supongamos ahora que queremos maximizar una función de k variables $f(x_1,...,x_k): R^k \rightarrow R$. Supongamos además que cada variable x_i puede tomar valores de un dominio $D_i = [a_i, b_i] \subseteq R$ y $f(x_1,...,x_k) > 0$ para todo $x_i \in D_i$.

Queremos optimizar la función f con una precisión prefijada: por ejemplo, supongamos que queremos d cifras decimales para los valores de las variables. Para conseguir dicha precisión, cada dominio D_i se divide en $(b_i - a_i)\cdot 10^d$ rangos de igual tamaño.

Denotaremos con m_i al menor entero tal que $(b_i - a_i)\cdot 10^d \leq 2^{m_i} - 1$. Entonces, una representación en la que cada variable x_i esté codificada como una cadena binaria de longitud m_i claramente satisface el requisito de precisión planteado.

Para interpretar una cadena, se aplica la siguiente fórmula:

$$x_i = a_i + decimal(cadena_2)\cdot\frac{b_i - a_i}{2^{m_i} - 1},$$

donde $decimal(cadena_2)$ representa el valor decimal de la cadena binaria a interpretar. Ahora cada cromosoma está representado por una cadena binaria de longitud $m = \sum_{i=1}^{k} m_i$; los primeros m_1 bits se mapean en un valor del rango $[a_1,b_1]$, el siguiente grupo de m_2 bits se mapean en un valor del rango $[a_2,b_2]$, y así sucesivamente; finalmente, el último grupo de m_k bits se mapean en un valor del rango $[a_k,b_k]$.

Para inicializar una población, podemos simplemente generar aleatoriamente tantos cromosomas como individuos deseamos que tenga la población. Sin embargo, en caso de que tengamos algún conocimiento acerca de la distribución de los óptimos potenciales, podemos usarla a la hora de crear las soluciones potenciales iniciales.

El resto del algoritmo es cíclico: en cada generación se evalúa cada cromosoma (usando la función f en las secuencias decodificadas de variables), se selecciona la nueva población respecto a la distribución de probabilidad basada en los valores de fitness (o aptitud), y se alteran los cromosomas de la nueva población mediante los operadores de mutación y cruce. Tras cierto número de generaciones, cuando no se observa mejora, el mejor cromosoma representa una solución óptima (posiblemente la global). A menudo se detiene el algoritmo tras un número fijo de iteraciones, dependiendo de criterios de velocidad y recursos.

Para el proceso de selección pueden emplearse diferentes métodos, aunque siempre se selecciona la nueva población respecto a la distribución de probabilidad basada en los valores de fitness. En el apartado 2.1.4 indicamos los operadores de selección más usados. El proceso de selección se ejecuta un número de veces igual al tamaño de la población, de modo que cada vez se selecciona un cromosoma simple para la nueva población. Los cromosomas pueden seleccionarse más de una vez, lo cual está de acuerdo con el Teorema de los Esquemas (del que hablaremos más adelante): los mejores cromosomas consiguen más copias, y los peores desaparecen.

Tras la selección se aplica el operador de cruce a los individuos de la nueva población. Uno de los parámetros del AG es la probabilidad de cruce p_c, que nos da el número esperado p_c ·tamaño_población de cromosomas que sufren la operación de cruce. Una vez seleccionados los individuos a cruzar, se combinan sus cromosomas y se sustituyen por dos hijos. Describimos las técnicas de cruce en el apartado 2.1.5.

El siguiente operador, la mutación, se aplica bit a bit. El parámetro de la probabilidad de mutación p_m nos proporciona el número de bits mutados esperado $p_m \cdot m \cdot$tamaño_población. Cada bit, en todos los cromosomas de la población total, tiene igual probabilidad de sufrir mutación. Los operadores de mutación se describen en el apartado 2.1.6.

Tras la selección, cruce y mutación, la nueva población se prepara para la siguiente evaluación. Esta evaluación de individuos (según la función de fitness f) se usa para construir la distribución de probabilidad para el siguiente proceso de selección. El resto de la evolución es simplemente una repetición cíclica de los pasos anteriores.

Puesto que es relativamente sencillo estar al tanto del mejor individuo en el proceso de evolución, en las implementaciones de AGs se acostumbra a almacenar el mejor individuo hasta el momento en una posición separada. De este modo, el algoritmo puede informar del mejor valor encontrado durante el proceso completo, en vez de contar sólo con el mejor valor de la población final.

2.1.2 Representación genética

La elección de la codificación y el diseño de la función de coste son dos puntos decisivos en la implementación del algoritmo genético. Se trata de codificar las soluciones al problema mediante cadenas de caracteres llamados cromosomas, aunque anteriormente se haya hablado siempre de cadenas de bits por ser la codificación más extendida.

Las características que debe cumplir una buena codificación son:

- Debe representar todo el estado de soluciones, o de lo contrario se estarán ignorando soluciones, entre las cuales puede estar precisamente la buscada.
- Debe asegurar que al aplicar los operadores de cruce y mutación no se generan individuos irreales, es decir, que no representan una verdadera solución al problema. Este problema se conoce como el problema del linkage, y volveremos a el más adelante, cuando se describa el operador de cruce.
- Debe cubrir todo el espacio de soluciones de modo continuo, o de lo contrario el proceso de búsqueda puede resultar aleatorio y por tanto poco eficaz.

El conjunto de caracteres usado en la representación se denomina alfabeto, y se representa por U. Cada uno de los valores del alfabeto es un alelo. La representación más básica consiste en utilizar cadenas de bits, es decir 0s y 1s, en cuyo caso tendríamos: $U = \{0,1\}$.

Ejemplo

Codificaremos en binario un ejemplo sencillo, en el que una Universidad tiene dos facultades, matemáticas y física, a los que han de incorporarse siete nuevos profesores. La codificación en este caso es muy elemental, puesto que cada profesor puede pertenecer a una de las dos facultades disponibles, de modo que basta con utilizar un único bit que indique la facultad a la que se une cada profesor: 0 para matemáticas y 1 para física. Por tanto, el cromosoma que represente una solución concreta de la asignación de profesores a facultades consistirá en una cadena de siete bits. Una posible solución sería:

0 0 0 1 1 0 1

que indicaría que los profesores primero, segundo, tercero y sexto ingresarían en la facultad de matemáticas, mientras que el cuarto, el quinto y el séptimo pertenecerían a la de físicas.

Si el problema se complica, puede que una codificación binaria no sea suficiente y que haya que ampliar el alfabeto con más cifras o con caracteres alfanuméricos. Supongamos que la Universidad considera ahora cinco profesores que deben elegir entre otras cuatro facultades: informática, telecomunicaciones, geología y biología. En este caso, las soluciones podrían representarse mediante cromosomas de cinco genes, cada uno de ellos con cuatro alelos. Asociando el alelo 1 a informática, el 2 a telecomunicaciones, el 3 a geología y el 4 a biología, tendríamos el alfabeto: $U = \{1,2,3,4\}$, y el cromosoma:

1 3 2 4 2

representaría que la asignación de los profesores es informática, geología, telecomunicaciones, biología y telecomunicaciones, respectivamente.
■

Si el problema lo requiere, se pueden emplear estructuras de datos más complejas para representar las soluciones, como por ejemplo matrices.

Como se ha dicho previamente, la forma en que se codifica el problema es un elemento clave para el buen funcionamiento del algoritmo. La mayoría de algoritmos genéticos usan una longitud de cromosoma fija y un orden fijo en las cadenas de bits. La codificación binaria es la más extendida, probablemente porque fue la primera que se usó. Holland dió una justificación para el uso de esta codificación: las codificaciones con un número pequeño de alelos y cadenas largas permiten un mayor grado de paralelismo implícito que las codificaciones con mayor número de alelos y cadenas cortas, puesto que una instancia del primer tipo de codificación contiene más instancias que una del segundo. Como contrapartida, la codificación binaria resulta poco natural para el ser humano, y para determinados problemas no es útil.

Problema del linkage y adaptación del código

La elección de un código fijo en longitud y orden tiene varios inconvenientes. En lo referente al orden, puede ocurrir que un orden fijo produzca la ruptura de buenos esquemas (codificaciones potencialmente buenas, ver apartado 2.1.8) debido a los efectos del cruce y la mutación. Este problema se denomina problema del linkage, y lo que deseable es que la funcionalidad relacionada con la posición tenga más probabilidad de permanecer junta bajo cruces y mutaciones. Una posible solución sería permitir que la codificación fuera cambiando el orden de los bits, pero no es fácil determinar qué posiciones han de conservarse en tiempo de compilación. De modo que la solución pasa por adaptar el código al tiempo que se avanza en la resolución del problema.

Otra razón para adaptar el código es que la longitud fija del cromosoma limita la complejidad de las soluciones candidatas.

A continuación se describen técnicas que resuelven estos dos problemas.

Inversión

Es un operador de reordenación, inspirado en la genética natural, en la que el valor de un gen es independiente de su posición en el cromosoma. De este modo, al invertir un cromosoma, se conserva mucha de la información que aparecía en el cromosoma original.

Para poder usar la inversión es necesario un mecanismo que permita interpretar la posición del alelo en el cromosoma de modo que su valor sea independiente de la posición en que aparece. Holland propuso que cada alelo llevara asociado un índice con su posición real, que se usaría en la evaluación del cromosoma. Por ejemplo, la cadena

0 0 0 1 0 1 0 1

se codificaría como

(1,0) (2,0) (3,0) (4,1) (5,0) (6,1) (7,0) (8,1)

donde el primer elemento de cada par representa la posición en el cromosoma original, y así este cromosoma es el mismo que

(1,0) (2,0) (6,1) (5,0) (4,1) (3,0) (7,0) (8,1)

El operador de inversión toma dos puntos de la cadena e invierte el orden de los bits entre ellos. Así, en el ejemplo anterior se han invertido los genes entre los bits 2 y 7. Este cambio no afecta al valor de fitness (valor de la función de coste) del cromosoma, y además ayuda a mantener los esquemas interesantes.

La idea que hay tras la inversión es producir órdenes entre los bits que beneficien esquemas con más probabilidad de sobrevivir (buenos esquemas).

2.1.3 Función de coste

Otro aspecto fundamental en el desarrollo de un AG es la elección de una buena función de coste. Esta función debe evaluar los individuos para indicar cuál es la calidad de la solución que representan, y poder realizar así el proceso de selección.

Ejemplo

Supongamos que tenemos una cadena de seis bits en la que queremos maximizar el número de unos. En este caso, la función de coste más indicada sería $f_1 = x$, donde x es el número de unos de la cadena (individuo) evaluada. Así, el individuo [0 0 0 0 1 1] tendría asociado un valor de la función de coste igual a 2, y el individuo [0 1 1 0 1 1] un valor de 4.
■

Normalmente los AGs tratan de maximizar una función, aunque también puede haber problemas de minimización. En este caso basta con utilizar la inversa de la función objetivo, o bien multiplicarla por -1.

Ejemplo

Retomando el ejemplo anterior, podría minimizarse el número de unos de las cadenas usando las funciones $f_2 = 1/x$ ó $f_3 = -x$. También podría modificarse el significado de la x para indicar el número de ceros de cada cromosoma, y usar así la función maximizadora f_1.

■

En ocasiones, los algoritmos genéticos tienen una función objetivo además de la función de coste. Es decir, que aunque la evaluación se realiza de acuerdo a una función, se trata de alcanzar un objetivo que se evalúa mediante otra función distinta. Esta función objetivo no tiene porqué ser una función numérica, sino simplemente un indicador de si se cumple o no cierto criterio. Esto suele ocurrir cuando se tratan problemas con restricciones o problemas en los que se pretende optimizar distintos parámetros conocidos como problemas multi-objetivo [53, 54, 55].

2.1.4 Operadores de Selección

Como ya hemos dicho, el operador de selección identifica a los mejores individuos de la población actual y los utiliza para generar la siguiente población. La selección debe asegurar que los mejores individuos (los de mejor valor de fitness) tienen mayor probabilidad de ser seleccionados como padres. Esta probabilidad se introduce para dejar abierto un camino a aquellas soluciones que no pertenecen a las mejores, para que también ellas puedan aportar información a la nueva generación.

La selección utilizada es muy importante para la correcta convergencia del algoritmo. La presión de selección debe ser tal que consiga un equilibrio entre explotación y exploración. Si la presión de selección es muy alta se corre el peligro de que individuos de la población inicial con fitness superior a la media, que representan óptimos locales pero no globales, se reproduzcan en exceso provocando una pérdida de diversidad y una convergencia prematura. En caso contrario, una presión de selección demasiado baja puede provocar una búsqueda aleatoria o una enorme ralentización del algoritmo. Probablemente, lo óptimo sería un operador de selección que evolucionara con el algoritmo, de modo que en las fases iniciales primara la exploración y cuya presión fuera creciendo hasta alcanzar un punto en el que primara la explotación.

Hay muchos métodos de selección. A continuación describiremos los más usados.

Selección por el método de la ruleta

La idea es dar a cada individuo una probabilidad de ser seleccionado acorde a su función de coste, y proporcional a su calidad dentro de la población evaluada. De este modo, cuanto mejor es su valor de fitness, mayor es la probabilidad de que sea seleccionado.

Para calcular la probabilidad de selección (P_{Si}) de cada individuo i, primero ha de evaluarse la función de coste para cada uno de ellos (FF_i), a continuación se obtiene la suma de todas ellas, obteniéndose P_{Si} según la siguiente expresión:

$$P_{si} = \frac{FF_i}{\sum_{i=1}^{n} FF_i}$$, donde n es el número de individuos de la población.

A continuación se calcula la probabilidad de selección acumulada (P_{ai}) para cada individuo, sumando para ello las probabilidades de selección de los individuos:

$$P_{ai} = \sum_{j=1}^{i} P_{sj}$$

Para seleccionar los individuos, se generan tantos números aleatorios (entre 0 y 1) como individuos se necesiten. Cada número aleatorio se compara con las probabilidades acumuladas y se escoge el individuo con una probabilidad asociada inmediatamente menor al número aleatorio generado. Al hacer los diagramas circulares correspondientes a las probabilidades acumuladas, se observa cómo cada individuo tiene una fracción proporcional a su probabilidad de selección.

Ejemplo

Volvamos al ejemplo de los cinco profesores repartidos en cuatro facultades. Supongamos que la asignación de cada persona a las diferentes facultades supone un coste para la universidad (en formación, traslado, papeleo, etc.) dado por la siguiente tabla:

Profesor	$C_{Informática}$	C_{Teleco}	$C_{Geología}$	$C_{Biología}$
1	22	35	12	16
2	6	24	15	18
3	14	16	32	30
4	18	23	5	2
5	12	26	15	26

Si se desea minimizar el coste de la nueva asignación de profesores, la función de coste a usar sería:

$$f(x) = \sum_{i=1}^{5} f_i(x)$$

siendo $f_i(x)$ el coste de cada empleado para la distribución (posible solución) evaluada. Así, una de las distribuciones óptimas sería la representada por el individuo:

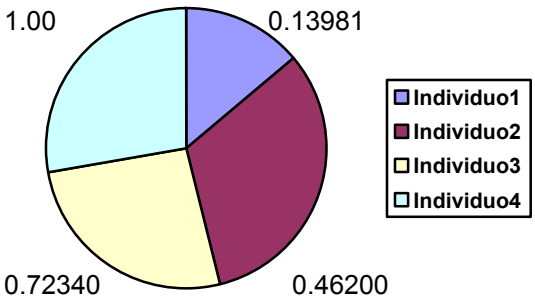
3 1 1 4 1

donde el primer profesor va a la facultad de geología, el segundo a la de informática, el tercero también a informática, el cuarto a biología y el último a informática.

En este caso, la función de coste y las probabilidades de selección y de selección acumulada para la siguiente población de cuatro individuos serían:

N	Individuo	FF_i	P_{Si}	P_{ai}
1	3 1 1 4 1	46	0.13981	0.13981
2	1 2 3 4 2	106	0.32218	0.46200
3	1 4 3 4 1	86	0.26139	0.72340
4	2 1 4 3 3	91	0.27659	1.00000

La representación de la probabilidad de selección acumulada es, por tanto:



Para cuatro números aleatorios generados, a continuación se muestra qué individuos se seleccionarían:

Nº aleatorio	N	Selección
0.23	1	3 1 1 4 1
0.07	4	2 1 4 3 3
0.97	3	1 4 3 4 1
0.65	2	1 2 3 4 2

■

Elitismo

Es un variante del modelo anterior, y consiste en guardar siempre el mejor individuo de la población para la siguiente población, normalmente sustituyéndolo por el peor. Hay estudios que aseguran que un algoritmo con selección elitista asegura la convergencia del AG hacia un óptimo global [56].

Selección basada en el ranking

Es un método muy empleado cuando la población evoluciona muy rápidamente y queda atrapada en un óptimo local. En este modelo los individuos se ordenan según su valor de fitness, y su probabilidad de selección depende del ranking.

Esta selección consigue disminuir la presión de selección y, por tanto, ralentizar la convergencia de la población. Sin embargo hay que estudiar bien la naturaleza del problema antes de aplicar este método, puesto que desprecia la información relativa a la función de coste absoluta, y en ocasiones esta información puede resultar necesaria.

Selección sigma (Forrest)

Se trata de una técnica que intenta adaptar la selección a medida que evoluciona el AG. En ella, el valor esperado de un individuo depende de su valor de fitness, del fitness medio de la población y de la desviación estándar de la población.

$$ExpVal(i,t) = \begin{cases} 1 + \dfrac{f(i) - \overline{f(t)}}{2 \cdot \sigma(t)} & si\ \sigma(t) \neq 0 \\ 1.0 & si\ \sigma(t) = 0 \end{cases}$$

donde ExpVal(i,t) es el valor esperado del individuo i en el instante de tiempo t, f(i) es el valor de fitness del individuo i, $\overline{f(t)}$ es el valor medio del valor de fitness de la población en el instante t, y $\sigma(t)$ es la desviación estándar de la función de coste de la población en el instante t.

Selección de Boltzman

La técnica anterior permite adaptar la selección de modo que se consiga una presión de selección constante a lo largo de la ejecución del AG. Sin embargo, en muchas ocasiones se desea que esta presión varíe. La selección de Boltzman funciona de modo similar al enfriamiento simulado [91], variando la temperatura que controla la presión de selección. La temperatura inicial debe ser elevada para que la presión de selección sea baja y, por tanto, se prime la exploración. La temperatura irá bajando gradualmente de modo que se incremente la presión de selección, primando entonces la explotación.

$$ExpVal(i,t) = \frac{e^{f(i)/T}}{\left\langle e^{f(i)/T} \right\rangle_t}$$

donde T es la temperatura y $\langle\rangle_t$ es el promedio de la población en el instante t.

Selección por torneo

Es el tipo de selección implementada en todos los prototipos, debido a que resulta muy apropiada para las características de la GPU.

Para cada iteración simple se toma cierto número s de individuos, llamado también tamaño de torneo, y se selecciona para la siguiente generación uno de los individuos del conjunto. Esta operación se repite tantas veces como individuos haya en una población, puesto que en cada selección producimos un solo individuo de la nueva población.

Es evidente que cuanto mayor sea el número de individuos s que se hacen competir cada vez, mayor es la presión de la selección para este método. Esto se debe a que el individuo más apto compite contra s - 1 individuos con peor aptitud de modo que, conforme crece el tamaño de torneo, la probabilidad de tomar una decisión incorrecta también crece (proporcionalmente). De hecho, se suele aproximar la probabilidad de tomar la decisión correcta (seleccionar el mejor individuo) con 1/s, aunque en realidad esta probabilidad es mayor.

El valor que se utiliza habitualmente es s = 2, y es el implementado en todos los prototipos desarrollados en el presente trabajo. En este caso, para determinar qué individuo es seleccionado de entre los dos candidatos, se genera un número aleatorio r entre 0 y 1: Si el número generado r es menor que un parámetro constante, se selecciona el mejor de los dos individuos, en otro caso se selecciona el peor. Después, los dos se devuelven a la población inicial para que puedan seleccionarse de nuevo.

El torneo implementado en este trabajo difiere un poco del anterior. En nuestro caso no existe una probabilidad de selección establecida por parámetro, sino que se hacen competir, en base a los valores de sus fitness, dos individuos seleccionados. Para ello, sacamos el fitness global de los dos individuos, y calculamos respecto a él la probabilidad de selección para cada uno de ellos. Si el primer individuo obtiene una probabilidad de selección *p*, entonces el segundo logra una probabilidad de *1 – p*. Si el número aleatorio generado es inferior a la probabilidad *p* del primer individuo, entonces se selecciona, si no seleccionamos el segundo. Por tanto, este tipo de torneo asigna a los individuos una probabilidad de selección variable (dependiendo de los individuos enfrentados) y en relación a su fitness, siendo el mejor de los rivales quien obtiene una probabilidad de selección mayor.

La selección de Boltzman se considera como una variante sofisticada de la selección por torneo, añadiendo enfriamiento simulado, en la que compiten dos individuos (i, j) y el ganador se determina según la fórmula:

$$\frac{1}{1+e^{\frac{f(i)-f(j)}{T}}}$$

donde T es la temperatura y f(i), f(j) son los valores de aptitud de los individuos.

Selección Steady-State

En este tipo de selección, solamente unos pocos individuos son reemplazados en cada generación, normalmente un pequeño número de individuos con los peores fitness.

2.1.5 Operadores de Cruce

El operador de cruce, junto con el operador de mutación, se utilizan para explorar el espacio de soluciones y poder encontrar soluciones buenas que están en otras zonas de este espacio. Ambos están inspirados también en la genética natural, en la reproducción y en la mutación azarosa de los genes, respectivamente.

El operador de cruce consiste en elegir aleatoriamente dos individuos progenitores de entre los seleccionados de la población, e intercambiar una parte de sus cromosomas dando lugar a dos nuevos individuos. De esta forma, los nuevos individuos tendrán parte de la información de diferentes individuos antiguos, perpetuando por una parte información de los padres, pero hallando al mismo tiempo nuevas soluciones al combinarla con la información de otro individuo.

El operador se aplica con una determinada probabilidad, y el intercambio del código de los cromosomas se realiza a partir de ciertos puntos que se seleccionan aleatoriamente. Aunque existen muchas variedades de cruce, lo más usual es seleccionar un único punto de cruce e intercambiar el código a partir de dicho punto. Esta técnica se denomina *cruce por un punto*.

Ejemplo

Supongamos que realizamos el cruce entre los dos primeros individuos de la tabla 2.2 (selección por el método de la ruleta):

3 1 1 4 1
1 2 3 4 2

Para aplicar el operador de cruce se genera un número aleatorio *mod(k)* siendo k el número de genes del cromosoma (en este caso k=5). Supongamos que se genera el 2; entonces intercambiamos los códigos de los individuos a partir del gen 2, generándose los dos siguientes individuos:

1 2 **1 4 1**
3 1 3 4 2

Como mencionamos anteriormente, es muy importante evitar el problema del linkage, es decir asegurar que el cruce no produce soluciones falsas. Por ejemplo, podemos suponer ahora el mismo problema pero ahora para 3 facultades y 4 profesores. Así, la codificación que podría usarse podría ser:

00 ➔ matemáticas
01 ➔ físicas
10 ➔ telecomunicaciones

donde se utilizan dos bits para el destino de cada profesor. Entonces, dos posibles soluciones podrían ser:

00 01 10 00
01 10 00 00

y si realizamos un cruce por el gen 3, obtendríamos los nuevos individuos:

01 1**1 10 00**
00 00 00 00

En la segunda solución obtenida, todos los profesores se asignan a la facultad de matemáticas y, por tanto, es una solución válida en principio. Sin embargo, en la primera solución el segundo profesor se asigna a una facultad de código 11, que no está definida. Este problema se subsanaría fácilmente utilizando una codificación entera en vez de binaria:

1 ➔ matemáticas
2 ➔ físicas
3 ➔ telecomunicaciones

Ahora, los individuos progenitores serían:

1 2 3 1
2 3 1 1

y, al cruzarlos por el gen 3, obtendríamos las dos soluciones válidas:

2 3 1 **1**
1 2 3 1

■

Para resolver los posibles errores de linkage al aplicar el operador de cruce, se puede optar por dos tipos de técnicas: las que se basan en la codificación o las que se basan en mecanismos reparadores. En [52] se describen las técnicas más utilizadas.

El operador de inversión en el cruce

Otra fuente de problemas a la hora de realizar el cruce, puede ser el uso del operador de inversión visto anteriormente.

Para identificar estos posibles problemas, veamos un ejemplo.

Ejemplo

Supongamos los siguientes individuos con cromosomas de 8 genes:

(1,0) (2,0) (6,1) (5,0) (4,1) (3,0) (7,0) (8,1)
(5,1) (2,0) (3,1) (4,1) (1,1) (8,1) (6,0) (7,0)

Si realizamos un cruce por el tercer gen, obtendríamos:

(1,0) (2,0) (6,1) (4,1) (1,1) (8,1) (6,0) (7,0)
(5,1) (2,0) (3,1) **(5,0) (4,1) (3,0) (7,0) (8,1)**

Lo que resulta en que el primer hijo tenga dos copias de los genes 1 y 6, y ninguna de los genes 3 y 5. Por tanto, el segundo hijo tiene dos copias de los genes 5 y 3, y ninguna de los genes 1 y 6.
■

Holland propuso dos soluciones para este problema:

- Permitir únicamente el cruce entre individuos cuyos cromosomas tengan la misma ordenación de genes. Es una solución, pero limita enormemente la capacidad de cruce, es decir, las situaciones en las que se puede aplicar - además, bajo cierta probabilidad - el operador.
- Aplicar un esquema maestro/esclavo, en el que la cadena maestro impone su ordenación de genes temporalmente a la cadena esclavo para realizar el cruce, devolviendo después a su orden original la cadena esclavo.

2.1.6 Operadores de Mutación

El operador de mutación introduce pequeñas alteraciones azarosas en el cromosoma con una probabilidad ínfima. Esto reestablece la diversidad que se va perdiendo con la aplicación sucesiva de los operadores de selección y cruce, permite la exploración de nuevas zonas del espacio de soluciones, y reduce la probabilidad de caer en óptimos locales.

Existen trabajos en los que se utilizan valores de probabilidad de mutación que varían con la progresión del algoritmo [57, 58]. La probabilidad de aplicación del operador suele ser baja, para evitar que el AG caiga en una búsqueda aleatoria. Sin embargo, en ocasiones puede ser necesario aumentar dicha probabilidad para recuperar la diversidad

de la población y evitar así la convergencia prematura. En esta línea, en [59] se ha diseñado un nuevo operador genético llamado *de regeneración,* basado en estas ideas, que ha dado buenos resultados.

El método más habitual de realizar la mutación es seleccionar al azar un gen del individuo y cambiar su alelo por otro de entre los disponibles en el alfabeto.

Ejemplo

Para ilustrar el mecanismo del operador de mutación, tomemos el primer individuo de la tabla 2.2:

3 1 1 4 1, donde el alfabeto era U = {1,2,3,4}.

Así pues, una mutación en el gen 3 podría resultar los siguientes individuos:

3 1 **2** 4 1
3 1 **3** 4 1
3 1 **4** 4 1

■

Al igual que ocurría con el cruce, hay que tener cuidado con el problema de linkage, y escoger una codificación que no produzca soluciones falsas cuando el operador de mutación se aplique.

2.1.7 Tamaño de la población

El tamaño de la población es un factor decisivo para lograr la convergencia de los AGs, puesto que el tiempo necesario para que un AG converja a una única solución depende del tamaño de la población. Goldberg y Deb publicaron un estudio [60] en el que demuestran que el tiempo para que un individuo se propague a toda la población utilizando los métodos más rápidos de selección es O(nlogn), siendo n el tamaño de la población. Aunque los AGs son eficientes, no son óptimos, es decir, no garantizan la obtención de una solución óptima.

Cuanto mayor sea el número de individuos, se explorarán más zonas del espacio de soluciones aunque, como contrapartida, el costo computacional será mayor. Dependiendo del tipo de problema y de los recursos disponibles, se deberá llegar a un compromiso entre el tamaño de la población y la calidad de las soluciones que se desea alcanzar.

2.1.8 Porqué funcionan los AGs: Base teórica

Aunque los AGs resultan intuitivos y fáciles de usar, describir e implementar, su formalización resulta compleja. Se ha realizado mucho trabajo en ese campo [51, 61, 62, 63, 64, 65, 66, 67, 68], y en este apartado se pretende recoger los conceptos más importantes.

La teoría fundamental que se esconde tras los algoritmos genéticos es que éstos trabajan descubriendo, favoreciendo y recombinando bloques de bits que aportan un alto valor a la función de coste de los individuos donde están presentes, y que se denominan Building Blocks (BBs). Este proceso se realiza de una manera altamente paralela. La idea es que las soluciones buenas tienden a estar compuestas por buenos BBs.

Para formalizar este concepto Holland introdujo el concepto de *esquema*, una plantilla que permite la exploración de similitudes entre cromosomas. En una codificación binaria, un esquema es una cadena de bits que se puede representar mediante una plantilla formada por unos (1), ceros (0) y comodines (*), que representan cualquier valor. Las cadenas H_i que pertenecen a un esquema H se llaman instancias de H.

Ejemplo

Sea el esquema: H = 1 * 0 * 1. Entonces, las siguientes cadenas son las instancias de H:

H_1 = 1 **0** 0 **0** 1
H_2 = 1 **0** 0 **1** 1
H_3 = 1 **1** 0 **0** 1
H_4 = 1 **1** 0 **1** 1

■

Un AG procesa un esquema del siguiente modo: Cualquier cadena de bits (valores en {0,1}) de longitud *long* es una instancia de 2^{long} esquemas (valores en {0,1,*}) diferentes. En consecuencia, una población de *n* individuos contiene instancias de entre 2^{long} y $n \cdot 2^{long}$ esquemas diferentes, puesto que puede haber individuos repetidos. Esto significa que, en una determinada generación, mientras el AG está evaluando explícitamente la función de coste de las n cadenas de bits que componen la población, en realidad está estimando implícitamente el valor medio de la función de coste de un número mucho mayor de esquemas. Aunque efectivamente los esquemas no están explícitamente representados y, por tanto, no son evaluados por el AG, el comportamiento en términos de aumento o disminución del número de instancias de los esquemas de la población sí se puede describir como si realmente se estuvieran calculando y almacenando los mencionados valores medios.

Ejemplo

Veamos que, efectivamente, a un cromosoma binario de *long* genes corresponden 2^{long} esquemas posibles:

- **long = 2**. Para una instancia cualquiera de 2 genes, obtenemos $2^2 = 4$ esquemas posibles:

1 0	1 0	1 *	* 0	* *

- **long = 3**. Para una instancia cualquiera de 3 genes, obtenemos $2^3 = 8$ esquemas posibles:

0 1 0	0 1 0	0 1 *	0 * 0	* 1 0	0 * *	* 1 *	* * 0	* * *

■

Los diferentes esquemas tienen diferentes características. Ya hemos visto que el número de comodines * en el esquema determina el número de cadenas que casan con el esquema. Hay dos importantes propiedades del esquema, el orden y la longitud definida; el Teorema de los Esquemas se formula sobre estas propiedades.

El *orden* de un esquema S (denotado por o(S)) es el número de posiciones con 0 ó 1, es decir de posiciones fijas (no comodines) en el esquema. En otras palabras, es la longitud del esquema menos el número de comodines. El orden defina la especialización de un esquema. Como veremos más adelante, la noción de orden del esquema es útil a la hora de calcular las probabilidades de supervivencia a las mutaciones del esquema.

Ejemplo

Sean los siguientes esquemas, todos de longitud 10:

$$S_1 = (* * * 0\ 0\ 1 * 1\ 1\ 0),\ S_2 = (* * * * 0\ 0 * * 0 *),\ S_3 = (1\ 1\ 1\ 0\ 1 * * 0\ 0\ 1)$$

Sus órdenes correspondientes son: $o(S_1) = 6$, $o(S_2) = 3$ y $o(S_3) = 8$. El esquema más específico es por tanto S_3.

■

La *longitud definida* de un esquema (denotada por $c(S)$) es la distancia entre la primera posición fija y la última. Define la compacidad de la información contenida en el esquema. Nótese pues que un esquema con una sola posición fija tiene una longitud definida de cero. Posteriormente veremos que el concepto de longitud definida de un esquema interviene en el cálculo de las probabilidades de supervivencia al cruce.

Ejemplo

Para los esquemas del ejemplo anterior, sus longitudes definidas son:

$$c(S_1) = 10 - 4 = 6,\ c(S_2) = 9 - 5 = 4,\ c(S_3) = 10 - 1 = 9$$

■

Recordamos que la simulación del proceso de evolución de los AGs consiste en cuatro pasos consecutivos que se repiten:

```
t ← t + 1
selecciona P(t) desde P(t - 1)
recombina P(t)
evalúa P(t)
```

El primer paso (`t ← t+1`) simplemente mueve el reloj de la evolución una unidad hacia delante. En el último paso (`evalúa P(t)`) simplemente se evalúa la población actual. El fenómeno principal del proceso evolutivo ocurre en los dos pasos intermedios del ciclo: la selección y la recombinación. Veremos ahora el efecto de estos dos pasos en el número esperado de esquemas representados en la población.

Sea H un esquema con al menos una instancia presente en la población en el instante de tiempo (generación) t. Sea $m(H,t)$ el número de instancias de H en la población del instante t, y $\hat{u}(H,t)$ el valor medio de la función de coste observado para las instancias de

H presentes en la población en el instante t. Para ver cómo evoluciona, debemos calcular el número de instancias de H esperado en el instante siguiente *t+1*:

$$E(m(H,t+1))$$

El número esperado de descendientes de una determinada solución x es:

$$f(x)/\bar{f}(t)$$

donde *f(x)* es el valor de la función de coste para el individuo x, y $\bar{f}(t)$ es el valor medio de la función de coste en el instante t.

Si se ignoran los efectos del cruce y la mutación, y suponemos que x es una instancia del esquema H:

$$E(m(H,t+1)) = \sum_{x \in H} \frac{f(x)}{\bar{f}(t)}$$

Que es equivalente a la expresión:

$$E(m(H,t+1)) = m(H,t)\frac{\hat{u}(H,t)}{\bar{f}(t)} \qquad (2.1)$$

Es decir, que aunque el AG no calcula $\hat{u}(H,t)$ explícitamente, la variación de las instancias de un determinado esquema de una población a la siguiente depende de esa cantidad: el número de cadenas en la población crece como la proporción del fitness del esquema respecto al fitness medio de la población. Esto significa que un esquema por encima del promedio recibe un incremento en el número de cadenas para la siguiente generación, mientras que un esquema inferior al promedio decrementa el número de cadenas, y un esquema medio permanece en el mismo nivel.

El efecto a largo plazo de la regla anterior es claro: si asumimos que un esquema H permanece por encima del promedio en un d%, es decir que $\hat{u}(H,t) = \bar{f}(t) + \in \cdot \bar{f}(t)$, entonces:

$$m(H,t) = m(H,0)(1+\in)^t \text{ y } \in = \frac{\hat{u}(H,t) - \bar{f}(t)}{\bar{f}(t)},$$

siendo d > 0 para esquemas por encima del promedio y d < 0 para esquemas por debajo del promedio.

Esta ecuación describe una progresión geométrica, de modo que no sólo sabemos que un esquema por encima del promedio recibe un incremento en el número de cadenas para la siguiente generación, sino que dicho esquema recibe un incremento *exponencial* en el número de cadenas en las siguientes generaciones. La intuición es que estos esquemas superiores definen una parte prometedora del espacio de búsqueda, y que se muestrean de una forma incremental exponencial. A la ecuación (2.1) se le llama ecuación del crecimiento reproductivo de esquemas.

Sin embargo, la selección sola no introduce ningún punto (o solución potencial) nuevo en el espacio de búsqueda: la selección simplemente copia algunas cadenas para formar una población intermedia. Por tanto, el siguiente paso del ciclo evolutivo (la recombinación) asume la responsabilidad de introducir nuevos individuos en la

población. Esto se logra mediante dos operadores genéticos: el cruce y la mutación, cuyos efectos veremos a continuación.

Operador de cruce

Supongamos que aplicamos el operador de cruce por un punto con una probabilidad p_c, y que se ha seleccionado como padre una instancia de un esquema H. Diremos que el esquema H ha *sobrevivido* al cruce si al menos uno de los dos descendientes también es instancia de H. Resulta evidente que la longitud definida de un esquema juega un papel significativo en la probabilidad de su destrucción o supervivencia.

Si H tiene una longitud definida c(H) en bits y el cromosoma tiene long bits, la probabilidad de escoger un bit del esquema para realizar el cruce, y por tanto destruir H es $c(H)/long-1$. En general, el lugar de cruce es seleccionado uniformemente entre (long – 1) posibles posiciones. Obviamente, la probabilidad de destruir H se obtiene multiplicando esta cantidad por la probabilidad de aplicar el operador, puesto que sólo algunos cromosomas sufren el cruce: $P(destruir\,H) = p_c \cdot \frac{\delta(H)}{long}$

Por tanto, la probabilidad $S_c(H)$ de que H sobreviva al cruce será, como mínimo:

$$S_c(H) \geq 1 - P(destruir\,H) = 1 - p_c \cdot \frac{\delta(H)}{long}$$

lo que indica que hay una mayor probabilidad de supervivencia para esquemas con menor $c(H)$, es decir, esquemas más cortos.

Nótese que la ecuación no es una igualdad exacta. Esto se debe a que incluso si el punto de cruce seleccionado está entre posiciones fijas del esquema, aún hay posibilidad (aunque pequeña) de que el esquema sobreviva.

Ejemplo

Sea el esquema H = (1 1 1 * * * * * 1 0), y las instancias de H: x = (1 1 1 0 0 1 0 0 1 0) e y = (1 1 1 1 0 1 1 1 1 0). Al cruzar ambos individuos el esquema H sobrevive si el punto de cruce está entre alguna de las posiciones fijas, como por ejemplo:

x' = (**1 1** 1 1 0 1 1 1 1 0)

y' = (1 1 **1 0 0 1 0 0 1 0**)

■

Así, el efecto combinado de la selección y el cruce nos da una nueva forma de la ecuación del crecimiento reproductivo de esquemas:

$$E(m(H,t+1)) = m(H,t)\frac{\hat{u}(H,t)}{\bar{f}(t)}\left[1 - p_c \cdot \frac{\delta(H)}{long}\right] \qquad (2.2)$$

Esta nueva versión de la ecuación nos revela el número esperado de cadenas que casan con el esquema H en la siguiente generación como una función del actual número de

cadenas que casan con el esquema, el fitness relativo del esquema, y su longitud definida. Resulta claro que los esquemas por encima del promedio con poca longitud definida serán muestreados en proporciones exponencialmente incrementales.

Operador de mutación

Consideremos ahora el siguiente operador, la mutación, que se aplica con una probabilidad de p_m. Este operador cambia aleatoriamente una posición simple del cromosoma de 0 a 1 o viceversa. Evidentemente, para que el esquema sobreviva a esta operación todas las posiciones fijas deben permanecer sin cambiar. Sólo las posiciones correspondientes a bits fijos son importantes: la mutación de al menos uno de estos genes destruiría el esquema. Por tanto, el número de bits importantes es precisamente el orden del esquema.

La probabilidad de que el bit i de H sea mutado es p_m y, por tanto, la probabilidad de que el bit sobreviva a la mutación es:

$$S_m(H_i) = 1 - p_m$$

Si el esquema H se compone de o(H) bits fijos, puesto que una mutación simple es independiente de otras mutaciones, la probabilidad de el esquema completo sobreviva es:

$$S_m(H) = \prod_{i=1} o(H) \cdot S_m(H_i) = (1 - p_m)^{o(H)}$$

Puesto que suele darse $p_m << 1$, es frecuente aproximar la anterior probabilidad como:

$$S_m(H) \approx 1 - o(H) \cdot p_m$$

El efecto combinado de la selección, el cruce y la mutación nos da la ecuación definitiva del crecimiento reproductivo de esquemas:

$$E(m(H,t+1)) \geq m(H,t) \frac{\hat{u}(H,t)}{\overline{f}(t)} \cdot (1 - p_m)^{o(H)} \cdot (1 - p_c \cdot \frac{\delta(H)}{long}) \quad (2.3)$$

Al igual que sus variantes más simples (2.1) y (2.2), esta ecuación nos indica que el número esperado de cadenas que se corresponden con el esquema H en la siguiente generación es una función del número de cadenas que actualmente casan con H, el fitness relativo del esquema, su longitud definida y su orden. A la vista de esta fórmula final, podemos concluir que los esquemas por encima del promedio con poca longitud definida y bajo orden serán muestreados con proporciones de incremento exponencial.

Nótese que la ecuación (2.3) asume que la función de fitness f devuelve sólo valores positivos. Cuando se aplican AGs a problemas de optimización en los que dicha función puede devolver valores negativos, debe realizarse algún mapeo adicional entre las funciones de optimización y fitness.

En resumen, la ecuación (2.1) mostraba el incremento de muestreo de los esquemas por encima del promedio, así como que dicho cambio es exponencial. Pero el muestreo por sí mismo no introduce ningún nuevo esquema (no representado en el muestreo inicial en

t = 0). Éste es el motivo por el que se introduce el operador de cruce, para permitir el intercambio estructurado aunque aleatorio de información. Adicionalmente, el operador de mutación introduce una gran variedad en la población. El efecto combinado de estos operadores en un esquema no es significativo si el esquema es corto y de bajo orden.

Teorema de los Esquemas

El resultado final de la ecuación de crecimiento (2.3) se conoce como el **Teorema de los Esquemas**, y muestra matemáticamente cómo varía un esquema de una generación a otra y, en consecuencia, cómo funciona un AG. Este teorema es la base fundamental.

En [21], este teorema se formula como:

"Los esquemas cortos, de orden bajo y por encima del promedio reciben muestras exponencialmente incrementales en generaciones subsiguientes de un algoritmo genético."

Un resultado inmediato de este teorema es que los AGs exploran el espacio de búsqueda mediante esquemas cortos y de orden bajo que, en consecuencia, son usados para el intercambio de información durante el cruce:

Hipótesis de los bloques de construcción (Buildings Blocks BBs) [21]

"Un AG intenta una ejecución cerca del óptimo a través de la yuxtaposición de esquemas cortos, de orden bajo y alta representación, llamados bloques de construcción."

Aunque algunos investigadores han probado estas hipótesis [46], para la mayoría de aplicaciones no triviales confiamos principalmente en los resultados empíricos. A lo largo de los últimos cincuenta años se han desarrollado muchas aplicaciones de los AGs que soportan la hipótesis de los BBs en muchos dominios diferentes. Sin embargo, esta hipótesis sugiere que el problema de la codificación en el AG es crítico para su ejecución, y que dicha codificación debe satisfacer la idea de los BBs cortos.

También se deduce que, al evaluar una población de n individuos, el AG está evaluando implícitamente los valores medios de coste de todos los esquemas presentes en la población, y aumentando o disminuyendo su presencia de acuerdo a esta ecuación.

Holland [25] demostró que, al menos, $tamaño\_población^3$ esquemas son procesados de forma útil, y esta propiedad se obtiene sin ningún requerimiento extra de memoria o procesado.

! *Recientemente, Bertoni y Dorigo [47] han demostrado que la estimación de $tamaño\_población^3$ es correcta sólo en el caso particular en que tamaño_población es proporcional a 2l, y dan un análisis de validez general.*

Esta constituye posiblemente el único ejemplo conocido de explosión combinatoria que trabaja para nuestro beneficio en vez de actuar como desventaja [21].

La evaluación simultánea e implícita de un gran número de esquemas en una población se conoce como *paralelismo implícito* de los AGs, lo que los hace especialmente indicados para la paralelización [47]. En el siguiente apartado se estudian las formas más usuales de realizar la paralelización.

Sin embargo, la hipótesis de los BBs es sólo un acto de fe que se viola en algunos problemas, como son los problemas defectivos, en los que algunos BBs (esquemas cortos de orden bajo) pueden desviar al AG, causando su convergencia a puntos subóptimos (óptimos locales).

El fenómeno defectivo está fuertemente relacionado con el concepto de *epístasis* que, en términos de AGs, significa una marcada interacción entre los genes de un cromosoma. Los genetistas usan este término para referirse al efecto de máscara o activación: un gen es *epistático* si su presencia suprime el efecto de un gen en otra posición. En otras palabras, la epístasis mide la extensión con la que la contribución al fitness de un gen depende de los valores de otros genes. Para un problema dado, un alto grado de epístasis significa que no pueden formarse BBs; por tanto, estamos frente a un problema defectivo.

Describiremos detalladamente el problema defectivo, así como posibles soluciones al mismo, en el apartado 2.3.

2.2 Algoritmos genéticos Paralelos

La computación paralela se ha convertido en una parte fundamental en todas las áreas de cálculo científico, ya que permite la mejora del rendimiento simplemente con la utilización de un mayor número de procesadores, memorias y la inclusión de elementos de comunicación que permitan a los procesadores trabajar conjuntamente para resolver un determinado problema [70].

Un programa es paralelo si en cualquier momento de su ejecución puede ejecutar más de un proceso. Un proceso es una copia de un programa o de una parte de él.

Al compartir la carga de trabajo entre N procesadores se puede esperar que el sistema trabaje N veces más rápido que con un solo procesador, lo que permite tratar problemas más grandes y complejos. Sin embargo, la realidad es diferente, ya que existen varios factores de sobrecarga que disminuyen el rendimiento previsto.

Para crear programas paralelos eficientes hay que poder crear, destruir y especificar procesos, así como la interacción entre ellos. Básicamente existen tres formas de paralelizar un programa [71]:

- *Paralelización de grano fino*: la paralelización del programa se realiza a nivel de instrucción.
- *Paralelización de grano medio*: los programas se paralelizan a nivel de bucle. Esta paralelización se realiza habitualmente de forma automática en los compiladores.
- *Paralelización de grano grueso*: basada en la descomposición del dominio de datos entre los procesadores, siendo cada uno de ellos el responsable de realizar los cálculos sobre sus datos locales.

En este trabajo se han empleado únicamente técnicas de grano grueso, como forma de aprovechar la capacidad de cálculo de la tarjeta gráfica para ejecutar algoritmos genéticos sobre una computadora.

La paralelización de grano grueso puede lograrse mediante tres estilos diferentes de programación:

- *Paralelismo en datos*: el compilador se encarga de la distribución de los datos, guiado por un conjunto de directivas que introduce el programador. Estas directivas hacen que, cuando se compila el programa, las funciones se distribuyan entre los procesadores disponibles. Este método resulta muy fácil de programar, pero como contrapartida suele tener una eficiencia inferior a la que se logra mediante paso de mensajes. Los lenguajes de paralelismo de datos más utilizados son el estándar HPF (High Performance Fortran) y el OpenMP [72, 73].
- *Programación por paso de mensajes*: es el método más utilizado para programar sistemas de memoris distribuida. En la forma más simple, los procesos coordinan sus actividades mediante el envío y la recepción de mensajes. Las principales ventajas que ofrece son la flexibilidad, la eficiencia, la portabilidad y la controlabilidad del programa. En su contra, el tiempo de desarrollo puede ser más elevado que para un paralelismo en datos. Las librerías más utilizadas son, por este orden, la estándar MPI (Message Passing Interface) [74] y PVM (Parallel Virtual Machine) [75].
- *Programación por paso de datos*: realiza la transferencia de datos entre los procesadores mediante primitivas unilaterales tipo put-get, lo que evita la necesidad de sincronizar emisor y receptor. Se trata de un modelo de programación de muy bajo nivel, pero muy eficiente. Actualmente, muy pocos fabricantes lo soportan.

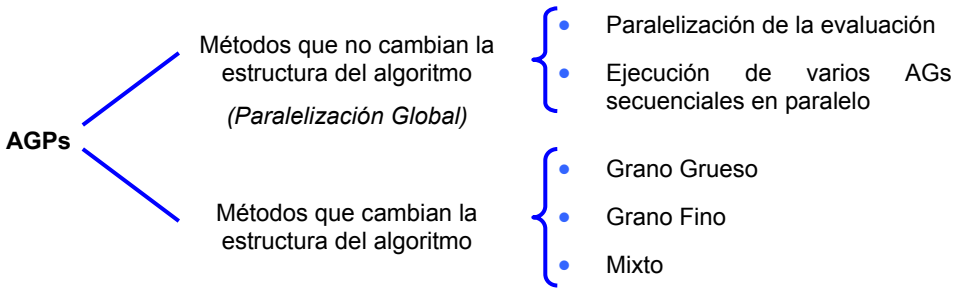
Los principales métodos de paralelización de algoritmos genéticos consisten en la división de la población en varias subpoblaciones (*demes*). Por ello, el tamaño y la distribución de la población entre los distintos procesadores será uno de los factores fundamentales a la hora de paralelizar un AG.

2.2.1 Clasificación de los AGs paralelos

Como se dijo anteriormente, el rendimiento real de los programas paralelos no alcanza el ideal por diversos factores. Puede ser que la estructura de los problemas no sea lo suficientemente regular como para obtener los rendimientos esperados. Otras veces los algoritmos y técnicas usadas, simplemente, no son fáciles de paralelizar.

Los AGs, sin embargo, tienen una estructura que se adapta perfectamente a la paralelización, puesto que se inspiran en un proceso intrínsicamente paralelo: la evolución natural, que trabaja utilizando muchos individuos al mismo tiempo.

Existen varias formas de paralelizar un AG [76]. A continuación se incluye un esquema de la clasificación de los AGPs (AGs paralelos):



La primera, y más intuitiva, es la global, que consiste básicamente en paralelizar la evaluación de los individuos manteniendo una población. Otra forma de paralelización global consiste en realizar una ejecución de distintos AGs secuenciales simultáneamente.

El resto de aproximaciones dividen la población en subpoblaciones que evolucionan por separado e intercambian individuos cada cierto número de generaciones. Si las poblaciones son pocas y grandes, como en nuestro caso, tenemos una paralelización de grano grueso [77]. Si el número de poblaciones es grande, y el número de individuos de éstas es reducido se trata de una paralelización de grano fino [78]. Por último, existen algoritmos que mezclan propiedades de estos dos últimos, y que se denominan mixtos.

Además de conseguir tiempos de ejecución menores, al paralelizar un AG estamos modificando el comportamiento del propio algoritmo, y esto hace que podamos encontrar otras soluciones y experimentar con las distintas posibilidades de implementación y los distintos factores que influyen en ella. Estas poblaciones van evolucionando por separado hasta detenerse en un momento determinado e intercambiar los mejores individuos entre ellas.

Técnicamente hay tres características importantes que influyen en la eficiencia de un AGP, y que se analizarán más a fondo en el apartado 2.2.3 [79]:

- La topología que define la comunicación entre las subpoblaciones.
- La proporción de intercambio, es decir el número de individuos a intercambiar.
- Los intervalos de migración, es decir la periodicidad con la que se intercambian los individuos.

2.2.2 Paralelización global

La paralelización global es la forma más sencilla de implementar un algoritmo genérico paralelo. Recordamos que consiste, bien en paralelizar la evaluación de los individuos, bien en realizar la ejecución simultánea de distintos AGs secuenciales. Además de sencilla, es una paralelización muy útil, puesto que permite obtener mejoras en el rendimiento con respecto al AG secuencial muy fácilmente, sin cambiar la estructura principal de éste.

Paralelización de la evaluación

En la mayoría de las aplicaciones de los AGs, la parte que consume un mayor tiempo de cálculo es la evaluación de la función de coste de los individuos. En estos casos se puede ahorrar mucho tiempo de cálculo simplemente encargando la evaluación de una parte de la población a distintos procesadores, de modo simultáneo.

Normalmente la evaluación de un individuo cuesta exactamente igual para todos ellos, de modo que el tiempo de cálculo puede disminuirse aproximadamente en N veces, siendo N el número de procesadores utilizados. Evidentemente habrá que tener en cuenta el costo de las comunicaciones: cuanto más sencilla sea la información a intercambiar, y menor la frecuencia de intercambio, menor será este costo. También

habrá que considerar el tipo de arquitectura que se esté utilizando y su facilidad para transmitir un tipo de datos u otro.

A continuación se incluye el esquema de un AG en el que se ha paralelizado la evaluación:

```
generación de la poblacion inicial
while (no se cumpla la condicion de parada) do
   do in parallel
      evaluacion de los individuos
   end parallel do
   seleccion
   produccion de nuevos individuos
   mutacion
end while
```

Este método mantiene una población única, y sólo la evaluación de los individuos se realiza en paralelo. La aplicación de los operadores (cruce y mutación) puede mantenerse global o realizarse también en paralelo, aunque generalmente el costo de comunicaciones necesario para paralelizar estas operaciones no compensa el tiempo de cómputo ahorrado.

Cuando el programa se para y espera el resultado de la evaluación para todos los individuos antes de proceder a realizar la selección necesaria para crear la siguiente generación, se dice que es una implementación síncrona. Este algoritmo tendrá las mismas características que su correspondiente secuencial.

Si, por el contrario, el procesador maestro no espera la llegada de todas las evaluaciones, tenemos una implementación asíncrona. En este caso, la estructura y comportamiento del algoritmo difiere de la versión secuencial, ya que la generación de los nuevos individuos no se realiza a partir de la misma información. Al no llegar las evaluaciones en el mismo instante de tiempo, determinados individuos pueden ser seleccionados en una generación anterior o posterior, e incluso no ser seleccionados. Esto puede deberse a que su evaluación llega tarde y para entonces la calidad de la nueva población ha aumentado, o simplemente a la propia probabilidad intrínseca a los AGs.

La mayoría de las implementaciones de AGPs globales son síncronas, debido a su facilidad de implementación.

Ejecución de varios AGs secuenciales en paralelo

La otra forma de paralelización global consiste únicamente en enviar varios AGs a distintos procesadores, y ver cuál es la mejor solución obtenida al final del proceso. El resultado es el mismo que si ejecutáramos dichos AGs secuenciales y escogiéramos la mejor solución de todas las obtenidas. El esquema en pseudocódigo es:

```
do in parallel
   generación de la población inicial
   while (no se cumpla la condición de parada) do
   evaluación de los individuos
      selección
      producción de nuevos individuos
      mutación
   end while
end parallel do
escoger la mejor solución
```

El modelo de paralelización global no hace ninguna distinción sobre la arquitectura del computador sobre el que se está ejecutando. Por tanto, se puede implementar tanto en un computador de memoria compartida como de memoria distribuida.

En un multiprocesador de memoria compartida, la población se puede guardar en la memoria compartida y cada uno de los procesadores puede leer los individuos que tiene asignados, evaluarlos y devolver los resultados, de tal forma que no se produzcan conflictos entre procesadores para acceder a la memoria, y no hace falta sincronización en este paso. Los problemas pueden aparecer únicamente como consecuencia de la propia red, y pueden hacer que disminuya la velocidad de ejecución del algoritmo.

El número de individuos asignado a cada procesador suele ser constante, pero en algunos casos puede ser necesario equilibrar la carga entre procesadores, para lo que puede utilizarse cualquier algoritmo dinámico diseñado para este propósito [71]. El equilibrio de la carga no es nada más que distribuir homogéneamente la cantidad de trabajo que realiza cada procesador de acuerdo a sus características.

En un computador de memoria distribuida la población se almacena normalmente en un procesador (maestro) que se encarga de enviar los individuos al resto de procesadores (esclavos), de recoger la información y de aplicar los operadores.

En cualquier caso, el costo de comunicaciones es similar para un multiprocesador de memoria distribuida y uno de memoria compartida. La diferencia estriba en que en un procesador de memoria distribuida se debe especificar explícitamente. Cuanto mayor es el número de esclavos utilizados, mayor es el costo de comunicaciones, pero evidentemente menor es el número de operaciones que debe realizar cada uno de los procesadores.

2.2.3 AGPs de Grano grueso

Las características fundamentales de un AGP de grano grueso son: la utilización de varias subpoblaciones relativamente grandes, y la migración o intercambio de individuos entre las distintas subpoblaciones [80, 81].

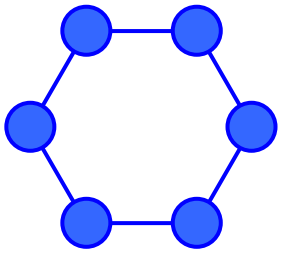
Éste es el tipo de paralelización más empleado, debido principalmente a las siguientes ventajas:

- La forma de implementar un AGP de grano grueso a partir de una versión secuencial es muy sencilla: Únicamente hay que tomar un conjunto de AGs secuenciales , ponerlos en varios procesadores y, cada cierto número de generaciones, intercambiar individuos. La mayoría del código de la versión secuencial queda exactamente igual después de paralelizar.
- La presencia de los computadores paralelos de grano grueso es habitual en l mayoría de los centros de investigación, y allí donde no están disponibles son fácilmente simulables mediante software como MPI (Message Passing Interface) o PVM (Parallel Virtual Machine).

Los AGPs de grano grueso también suelen llamarse AGs distribuidos, debido a que se suelen implementar sobre máquinas de memoria distribuida. En ocasiones se les llama

AGP de "isla" (*island model*), ya que consisten en un modelo de poblaciones en el que las subpoblaciones están relativamente aisladas. En la siguiente figura se incluye un diagrama en forma de grafo de un AGP de grano grueso.

En él se representan las poblaciones por nodos, y las aristas representan líneas de intercambio de individuos al realizar la migración:



En principio, podría parecer que, dado que utilizamos subpoblaciones de menor tamaño que en un AG secuencial, un AGP de grano grueso debería converger en un número menor de generaciones. Sin embargo, aunque es cierto que con una población menor un AG converge más rápido, también es cierto que la calidad de la solución no tiene porqué ser la misma si el problema tratado es complejo.

Los AGP de grano grueso simulan el aislamiento geográfico de las diferentes civilizaciones, y el intercambio esporádico de características que se realiza con la emigración.

A continuación se puede ver un esquema en pseudocódigo de un AGP de grano grueso, en el que la *frecuencia* es el número de generaciones que pasa entre dos intercambios de individuos:

```
inicializar P subpoblaciones con N individuos cada una
Numero de generacion = 1
while (no se cumpla la condicion de fin) do
   for (cada subpoblacion) do in parallel
      evaluar y seleccionar individuos por su funcion de coste
      if (Numero de generacion mod frecuencia) = 0 then
         enviar K<N mejores individuos a poblacion vecina
         recibir K mejores individuos de poblacion vecina
         reemplazar K individuos de la poblacion
      end if
      producir nuevos individuos
      aplicar operador de mutacion
   end parallel do
   Numero de generacion++
end while
```

Como se indicó anteriormente, hay tres parámetros importantes que influyen en la eficiencia de un AGP:

- La topología que define la comunicación entre las subpoblaciones.
- La proporción de intercambio, es decir el número de individuos a intercambiar.
- Los intervalos de migración, es decir la periodicidad con la que se intercambian los individuos.

En los siguientes apartados analizaremos la influencia de cada uno de estos factores en el funcionamiento de un AGP de grano grueso, y las distintas alternativas que se presentan.

Topologías de comunicación

La topología es un factor fundamental en el rendimiento de un AGP, puesto que determina la velocidad con que una buena solución se propaga de una subpoblación al resto.

Si la topología tiene muchas conexiones entre las subpoblaciones, las buenas soluciones se transmiten rápidamente de una población a otra. Es evidente que también lo harán las malas soluciones, pero precisamente el proceso de selección - gobernado por la función de coste - controla este esparcimiento, limitándolo a la pura probabilidad de selección que pueda tener una mala solución.

Por el contrario, si las poblaciones tienen poca comunicación entre ellas, las soluciones se extienden más lentamente, permitiendo la aparición de varias soluciones, así como una evolución más aislada de cada grupo. Estas distintas soluciones se pueden utilizar posteriormente para generar individuos que superen a los obtenidos mediante una convergencia más homogénea.

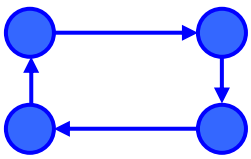
Otro factor en el que interviene la topología es el coste de las comunicaciones. Aunque normalmente no se realizan tantos intercambios como para ralentizar el AGP, es necesario buscar un compromiso entre la topología elegida y el costo de las comunicaciones, para no perder las ventajas obtenidas sobre el rendimiento con la paralelización.

La forma general es utilizar una topología estática que se mantiene constante a lo largo de toda la ejecución del algoritmo. Muchas de estas topologías estáticas aprovechan la propia topología de la red de comunicaciones entre procesadores.

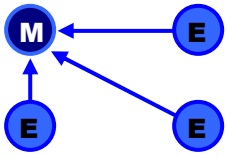
Otra opción es la implementación de una topología dinámica. En este tipo, el intercambio entre subpoblaciones se realiza entre procesadores distintos cada vez, en función de un criterio que suponga una mejora para el algoritmo. Uno de los factores que más se tiene en cuenta para este tipo de AGP es la diversidad de la población, es decir, se trata de favorecer el intercambio con aquellas subpoblaciones con un mayor número de individuos iguales.

Tres de las topologías más usadas en la implementación del modelo de islas son:

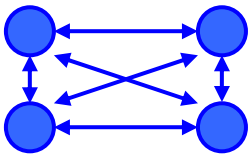
- Topología en anillo: las poblaciones están distribuidas en un anillo, y sólo hay intercambio de individuos entre vecinos.



- Topología maestro-esclavo: todos los procesadores esclavos intercambian sus mejores individuos con el maestro.



- Comunicación todos con todos (all-to-all): todos los procesadores intercambian información con cada uno de los otros.



Proporción y frecuencia de intercambio

Tanto la frecuencia como la proporción de intercambio son muy importantes para la convergencia del algoritmo y para la calidad de las poblaciones. Aunque en principio se puede suponer que cuanto mayor sea el intercambio mejor se propagan las buenas soluciones, esto no es cierto totalmente, ya que puede suceder que el excesivo intercambio de individuos entre las poblaciones convierta el AGP en una búsqueda prácticamente aleatoria, al no permitir que el algoritmo se desarrolle con normalidad.

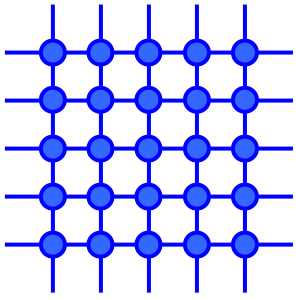
Algunas implementaciones realizan el proceso de migración únicamente cuando las poblaciones han convergido totalmente [82, 69]. Con ello se trata de introducir una diversidad en las poblaciones, y aliviar así problemas de convergencia prematura. Otras aproximaciones utilizan distintas políticas de intercambio y lo realizan después de un determinado número de generaciones, o bien con una periodicidad fijada a priori, y que se mantiene constante a lo largo de toda la ejecución del programa.

Si la migración se produce muy pronto, puede suceder que las propiedades de los individuos intercambiados sean débiles e influyan muy levemente en el proceso de búsqueda.

Por todo lo dicho anteriormente, queda patente que es conveniente realizar un estudio de distintas políticas de migración, teniendo en cuenta a la vez la topología de las poblaciones, al afrontar un problema concreto. En nuestro caso, este estudio constituye una línea de trabajo que dejamos abierta para el futuro.

2.2.4 AGPs de Grano fino

Los algoritmos genéticos de grano fino se conocen también como AGP *grid* o *en parrilla*, debido a la disposición de las poblaciones sobre los procesadores: Los individuos se disponen en una parrilla de dos dimensiones, con un individuo en cada una de las posiciones de la rejilla [52, 50]:



La evaluación se realiza simultáneamente para todos los individuos, y la selección, reproducción y cruce se realizan de forma local con un número reducido de vecinos. Con el tiempo se van formando grupos de individuos genéticamente homogéneos, como resultado de la lenta difusión de individuos. A este fenómeno se le llama *aislamiento por distancia*, y es debido a que la probabilidad de interacción entre los individuos disminuye con la distancia.

Este tipo de implementación simula las relaciones personales entre individuos de una misma localidad. Es decir, normalmente dos individuos que vivan cerca tienen más probabilidad de relacionarse que dos que vivan más separados.

El esquema en pseudocódigo de un AGP de grano fino es:

```
for (cada punto de la parrilla)
   do in parallel
      generar un individuo aleatoriamente
   end parallel do
   while (no se cumpla la condicion de fin) do
      for (cada punto de la parrilla k)
         do in parallel
            evaluate individual in k
            seleccionar un individuo vecino q
            producir descendiente de k y q
            asignar uno a k
            aplicar operador mutacion sobre k con probabilidad Pm
         end parallel do
   end while
```

En la descripción anterior, los vecinos que se consideran son generalmente los cuatro u ocho más próximos. La forma de seleccionar al individuo de la vecindad con el que se interactúa se puede hacer de varias formas, aunque la más utilizada es por torneo, es decir que compiten entre ellos mediante el valor de su función de coste.

De igual forma, la sustitución de los antiguos individuos por los nuevos a partir de los descendentes se puede hacer eligiendo el de mejor función de coste, o mediante un proceso aleatorio.

Todos estos procesos se pueden hacer de una manera dinámica, o bien realizarse con individuos que viajen, o cualquier otra variante que se desee implementar.

Los modelos de grano fino se adaptan mejor a las máquinas de tipo SIMD, ya que las operaciones necesarias para las comunicaciones locales son muy eficaces implementadas sobre este hardware.

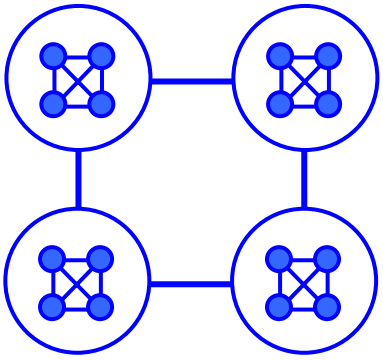
2.2.5 AGPs Híbridos

Un algoritmo genético híbrido es una combinación de AGPs de grano fino con los de grano grueso. Algunos de estos algoritmos híbridos añaden un nuevo grado de complejidad al entorno de los AGP, pero otros utilizan la misma complejidad que uno de sus componentes.

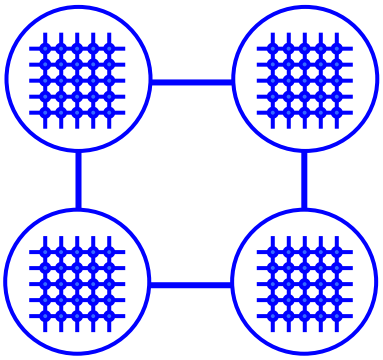
Cuando se combinan dos métodos de AGP, éstos forman una jerarquía. Los AGP híbridos son normalmente paralelos de grano grueso en el nivel superior. Algunos de

ellos tienen un AGP de grano fino en el nivel inferior, y otros tienen AGP de grano grueso en ambos niveles.

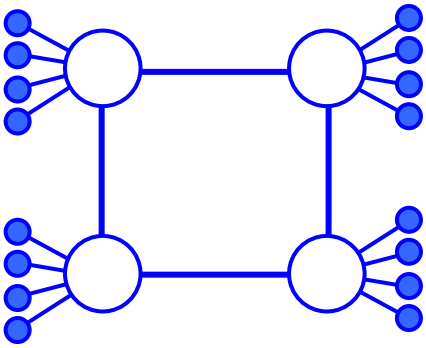
Puesto que en este trabajo no se han implementado AGPs híbridos, no nos detendremos más en este tipo de algoritmos. A continuación se incluyen diferentes esquemas de AGP híbridos que utilizan distintas jerarquías, a modo de curiosidad:



AGP híbrido que combina un AGP de grano grueso tanto en el primer nivel de la jerarquía como en el segundo



AGP híbrido que combina un AGP de grano grueso en el primer nivel, con uno de grano fino en el segundo



AGP híbrido que combina un AGP de grano grueso con un AGP global

2.3 Funciones defectivas (deceptive functions)

El término *deceptive* - que nosotros traduciremos como *defectivo* - fue introducido por [18] para probar las limitaciones de los AGs.

Aunque, como ya hemos visto, los AGs son algoritmos muy potentes que resuelven con éxito muchos problemas de optimización complejos, algunos otros resultan difíciles de resolver y se denominan problemas *GA-Hard*. Los problemas defectivos son una clase particular de estos problemas, y explotan la debilidad de la codificación de los cromosomas [1], [2]. Chow propone un método basado en algoritmos genéticos [19] que resuelve los problemas GA-Hard, desarrollando un mapeo del genotipo al fenotipo que permite capturar también información de esquemas.

2.3.1 Teoría de los esquemas

Antes de definir los problemas defectivos, revisaremos la teoría básica de los esquemas a fin de poder entender mejor la naturaleza de estos problemas de optimización.

El concepto de esquemas intenta explicar la habilidad que tienen los AGs para desarrollar una búsqueda global en el espacio del problema, generalmente extenso y con gran número de dimensiones [3]. Como vimos anteriormente, trabajamos con la hipótesis de la existencia de bloques de construcción (BBs, Building Blocks) en los cromosomas, a los que precisamente llamamos *esquemas*. Recordamos que un esquema es un patrón de cromosoma (de una parte del cromosoma) constituido por símbolos definidos (0 y 1) y por un símbolo comodín (*) que casa con 0 y con 1 indistintamente.

Así, mientras que un cromosoma (por ejemplo el 10010011) representa un punto simple en el espacio de soluciones, un esquema del mismo número de bits representa un hiperplano en el espacio de soluciones. Por ejemplo, el esquema 10*1***1 (que casa con el cromosoma anterior) forma un hiperplano consistente en 16 puntos de datos, puesto que cada comodín * puede ser un 1 ó un 0 ($4^2 = 16$).

Se dice que un esquema *muestrea* su hiperplano asociado porque, cuando sobrevive de generación en generación, los puntos de datos de su hiperplano son probados repetidamente como posibles buenas soluciones. Así, a través de la recombinación, mutación y selección genéticas, los hiperplanos prometedores en el espacio de soluciones reciben proporciones de muestreo mayores, gracias a un incremento exponencial de las cantidades del esquema en la población de cromosomas.

Genotipo VS fenotipo

Un cromosoma binario puede consistir en fragmentos de secuencias binarias, cada uno de los cuales se llama *gen*, que corresponden a un parámetro de la función de optimización. Normalmente, se escoge un esquema particular de codificación de números binarios para trasladar cada secuencia binaria al correspondiente valor

numérico (u otro tipo de información) del parámetro. Por ejemplo, podemos usar el complemento a dos para traducir desde binario a entero.

El cromosoma binario representa el *genotipo*, mientras que la lista de parámetros decodificados representa el *fenotipo*.

2.3.2 Problemas de fondo

A continuación comentamos algunos de los problemas que subyacen al uso de este tipo de funciones de aptitud.

Métodos de codificación binaria

Plantearemos aquí el problema de la codificación cuando se traduce un parámetro del fenotipo a un número binario. El método de codificación más sencillo y usado es el complemento a dos. El problema de este tipo de codificaciones es que se pueden producir discontinuidades en el proceso de búsqueda, debido a que un paso simple en el espacio fenotípico (por ejemplo de 15 a 16) requeriría múltiples pasos en el espacio genotípico (5 pasos desde 00001111 hasta 00010000).

Se hicieron varios intentos, usando métodos de codificación alternativos como por ejemplo el código Gray, para solucionar este problema ([4], [5]). La solución parecía pasar por emplear códigos tales que la distancia Hamming entre dos números (valores del fenotipo) consecutivos fuera uno.

Sin embargo, incluso usando códigos como el Gray, un espacio de problema puede seguir resultando complejo para la búsqueda que realizan los AGs.

Problemas defectivos

Como dijimos anteriormente, algunos de los problemas que resultan *duros* para los AGs son los llamados problemas defectivos (*deceptive problems*). Se describe y discute detalladamente este tipo de problemas en [1], [2] y [7].

En primer lugar veremos qué tipos de esquemas pueden llevar a engaño (*deception*) [19]. Ya sabemos que el fitness de un esquema determinado depende de los valores de fitness de sus puntos muestreados (los que forman su hiperplano). Y, por la naturaleza de los AGs, los esquemas con altos valores de fitness tienen mayor probabilidad de sobrevivir y pasar a la siguiente generación. A continuación daremos algunas definiciones útiles para el posterior planteamiento de los problemas defectivos:

- Un esquema con n bits definidos (1s ó 0s) se dice un esquema ***de orden n***. Por ejemplo, los esquemas *0**0, *0**1, *1**0 y *1**1 son todos esquemas de orden dos.
- Los esquemas con igual orden que, además, tienen definidos los bits de las mismas posiciones son ***competidores primarios*** en el proceso de selección, puesto que dichos esquemas compiten como planos ortogonales en el hiperespacio. Un hiperplano de orden inferior también contiene una colección de hiperplanos de

orden más alto, que comparten bits definidos con el hiperplano de orden inferior. Por ejemplo, el hiperplano de orden 2 0***0 contiene a su vez hiperplanos de orden 3 tales como 00**0 y 01**0 (entre otros), mientras que el esquema 1***0 contiene otro conjunto de hiperplanos de orden 3 tales como 10**0 y 11**0 (entre otros).

- Un esquema de orden n y los esquemas de orden k (donde n < k) que contiene se dicen ***relevantes*** uno respecto del otro. Así, cuando el esquema de menor orden compite en un proceso de selección, todos sus esquemas relevantes de nivel superior también participan en la competición. Por ejemplo, la competición entre los esquemas de orden 2 0***0, 0***1, 1***0 y 1***1 también implica las competiciones entre los esquemas de orden 3 00**0, 01**0, 00**1, 01**1, 10**0, 11**0, 10**1 y 11**1.

  Recíprocamente, las competiciones entre los hiperplanos de orden superior también implican la competición de sus hiperplanos relevantes de orden inferior.

Una vez expuestos estos conceptos, podemos definir cuándo se produce una situación defectiva [19]: cuando la competición entre hiperplanos de un nivel superior k se encamina a una solución global que es radicalmente diferente, en términos de distancia de Hamming, de la solución global de las competiciones entre los hiperplanos relevantes de orden n, donde n < k.

Ejemplo

Sean los siguientes valores de fitness para la función defectiva f1:

f1(000)	28	f1(001)	26
f1(010)	22	f1(100)	14
f1(110)	0	f1(011)	0
f1(101)	0	f1(111)	**30**

Valores de la función f1

Con la función f1, la solución global de una competición de hiperplanos de orden 3 es 111 (fitness 30).

Ahora consideremos el hiperplano relevante de 111 representado por *11. Cuando compite con *00, *01 y *10 en competiciones de orden 2, el fitness promedio de *11 es 15 ((0 + 30) / 2), en contraposición a los valores promedio de *00 ((28 + 14) / 2 = 21), de *01 ((26 + 0) / 2 = 13) y de *10 ((22 + 0) / 2 = 11). En función de estos valores, el esquema *00 parece el que más probablemente ganará la competición de hiperplanos de orden 2.

La diferencia, en distancia de Hamming, entre las dos soluciones globales 111 y *00 provoca una situación defectiva (*deception*).

■

Una vez vistos los conceptos anteriores, podemos dar al fin una definición de problema defectivo (*deceptive problem*) [19]:

> *Un **problema defectivo** es cualquier problema de un orden k determinado que implica engaño (deception) en una o más competiciones de hiperplanos relevantes de orden n, donde n < k.*

Un problema defectivo desvía a los AGs a converger incorrectamente a una región en el espacio del problema llamado *atractor defectivo* (óptimo local).

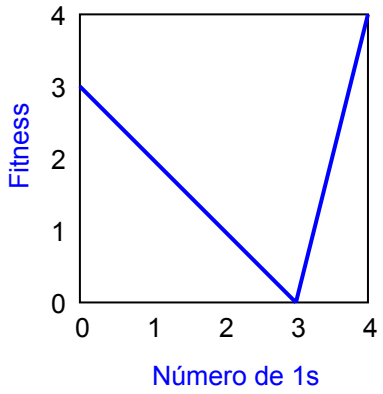
2.3.3 Funciones trampa (trap functions)

Un ejemplo de las funciones defectivas son las llamadas funciones *trampa* (*trap functions*).

> *Una **función trampa** es una función del número de 1s de una cadena de bits. Es una función defectiva porque desvía la búsqueda hacia una solución que está muy alejada de la correcta (óptimo global) [20].*

Ejemplo

La función mostrada en la figura es una función trampa, puesto que hay una tendencia hacia la solución 0000 (con fitness 3), que está máximamente alejada de la mejor solución 1111 (con fitness 4).



El problema se desvía porque los pequeños cambios incrementales en la solución son recompensados, dirigiendo erróneamente la búsqueda hacia el atractor defectivo.

■

Para definir formalmente una función trampa, introduciremos primero la definición de otra función auxiliar. Sea $\mathbf{x} = (x_1,...,x_l)$ una cadena binaria de longitud l. La función $u(\mathbf{x})$ de $\mathbf{x}$ se define como:

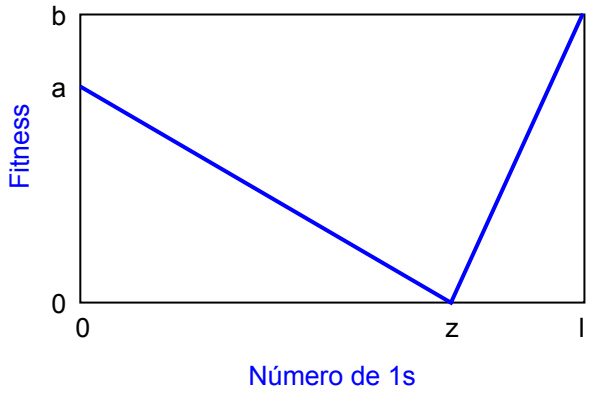
$$u(\mathbf{x}) = u(x_1,...,x_l) = x_1 + ... + x_l = \sum_{i=1}^{l} x_i$$

Una función trampa $f(\mathbf{x})$ se define, basándose en $u(\mathbf{x})$, del siguiente modo:

$$f(\mathbf{x}) = F(u(\mathbf{x})) = \begin{cases} \frac{a}{z}(z - u(\mathbf{x})), \text{ si } u(\mathbf{x}) \le z \\ \frac{b}{l-z}(u(\mathbf{x}) - z), \text{ eoc} \end{cases}$$

donde *a* es el óptimo local (posiblemente defectivo), *b* es el óptimo global, y *z* es la posición del cambio de inclinación que separa la cuenca de atracción de los dos óptimos. Los valores de los parámetros *a*, *b* y *z* de la función trampa determinan el grado de dificultad que supondrá para el AG encontrar el óptimo global *b* en contraposición al óptimo local *a*.

La forma general de una función trampa de l bits es:



2.3.4 Algunos intentos para resolver problemas defectivos

Se han propuesto diferentes técnicas para afrontar los problemas defectivos [19]. Algunas de ellas se centran en métodos de codificación. Como comentamos en apartados anteriores, puede adjuntarse a cada bit binario del cromosoma un bit etiqueta que especifica la localización en el fenotipo [1], [8]. Así, una cadena binaria cualquiera 11011001 se representa como una lista de pares ordenados valor-posición:

(1 1) (2 1) (3 0) (4 1) (5 1) (6 0) (7 0) (8 1)

Mediante esta codificación, el orden de los pares ordenados puede modificarse pero, a la hora de cruzar dos individuos, se alinean previamente sus cromosomas en el mismo orden.

El problema de estas aproximaciones es que implican un espacio de búsqueda adicional para encontrar la permutación correcta de ordenación de los bits, cuyo tamaño es similar al espacio de optimización de la función original. En [12] se sugiere una reordenación de las posiciones de los genes, en lugar de las posiciones de los bits, para preservar los bloques de construcción (BBs).

En [9] se propone un AG de dos niveles: cada gen del nivel inferior del cromosoma se etiqueta con un gen de control de alto nivel que lo activa o desactiva. Esta aproximación es muy similar a la de genes redundantes, descrita en [10], donde múltiples bits genotípicos deciden la activación o desactivación de un bit fenotípico.

En [1] y [11] se mantienen varias poblaciones y se permite migración entre diferentes subpoblaciones, a fin de mantener la diversidad. En la medida en que al menos una subpoblación no converja hacia el atractor defectivo, existe alguna posibilidad de alcanzar el óptimo global.

2.3.5 Un método de mapeo evolutivo

En [19] se propone un método basado en el modelo de AG con una población simple, aunque puede extenderse fácilmente a múltiples poblaciones.

En lugar de una población de cromosomas, se mantiene una población de celdas, es decir que cada individuo consiste en una de estas celdas. Cada celda contiene dos cromosomas: un *cromosoma binario de datos*, que almacena el genotipo para la función de optimización, y un *cromosoma de mapeo*, que almacena las posiciones de los bits en forma de enteros.

Ejemplo

Una celda inicial podría tener los dos siguientes cromosomas:

1	1	0	1	1	1	0	0

0	1	2	3	4	5	6	7

El primer cromosoma es el de los datos, y almacena los bits del genotipo, y el segundo es el cromosoma de mapeo, y define el mapeo del genotipo al fenotipo, es decir, el segundo cromosoma indica la posición de los genes del primero.

El cromosoma de mapeo del ejemplo mapea el genotipo a un fenotipo (1, 1, 0, 1, 1, 1, 0, 0) exactamente igual al propio genotipo. Otro cromosoma de mapeo producirá un fenotipo diferente. Así, el cromosoma de mapeo:

7	6	0	1	2	3	4	5

mapea el cromosoma de datos al fenotipo (0, 0, 1, 1, 0, 1, 1, 1), que puede ser un parámetro de una función de optimización.

■

En el proceso de evolución, los cromosomas de datos y mapeo sufren operaciones genéticas independientes. Los cromosomas de mapeo atraviesan además operaciones de permutación adicionales, que alteran la ordenación de los bits.

Mientras que las operaciones genéticas que se aplican a los cromosomas de datos son las tradicionales ([19] sugiere el cruce por dos puntos y la mutación habitual), las diferentes implementaciones de los operadores genéticos para cromosomas de mapeo producen salidas significativamente distintas.

Si la ordenación de bits del cromosoma de mapeo se mantiene constantemente como permutación de las posiciones de bits originales, la aproximación es muy similar a la que usaba el bit etiqueta, citada anteriormente. Sin embargo, este método de permutación por sí mismo no ha demostrado ser muy eficiente afrontando problemas defectivos [19].

En [19] se sugiere un desacoplamiento del genotipo y del proceso de mapeo, permitiendo para ello que los cromosomas de mapeo se crucen y muten libremente usando operadores genético enteros, sin mantener un mapeo uno-a-uno entre un bit genotípico y uno fenotípico.

Operadores genéticos

Como se ha indicado, los operadores que se aplican a los cromosomas de datos son los habituales en AGs. Para los cromosomas de mapeo, [19] sugiere operadores genéticos especiales.

El operador de cruce que se emplea es un operador tradicional entero de cruce por dos puntos.

El operador de mutación es un operador entero que altera aleatoriamente un gen en una de la posiciones de bits. Además, se incluye una operación de reemplazamiento de genes en el proceso de mutación. A través de esta operación, un gen del cromosoma de mapeo puede ser copiado a otro gen. El proceso completo de la operación genética romperá el mapeo uno-a-uno entre un bit genotípico y uno fenotípico. Un bit genotípico del cromosoma de datos puede ser mapeado a más de un bit en el fenotipo.

Ejemplo

Dado un cromosoma de datos (1,0,0,1,1,1,0,0) y un cromosoma de mapeo (1,4,0,1,2,7,4,5), se puede construir el fenotipo (0,1,1,0,0,0,1,1). En dicho fenotipo, las posiciones de bits 3 y 6 del cromosoma de datos no se usan.

■

Los fundamentos de esta aproximación son:

Se fuerza a los cromosomas de datos a concentrarse en la exploración y supervivencia en el espacio genotípico donde los materiales genómicos sufren constantemente construcción y destrucción de esquemas. Por otra parte, los cromosomas de mapeo se concentran en el esfuerzo de obtener un mapeo óptimo entre el genotipo y el fenotipo.

Con celdas de múltiples cromosomas pueden desarrollarse diferentes modos de mapeo al mismo tiempo, puesto que cada mapeo se asocia a un genotipo particular. Por el contrario, algunas aproximaciones anteriores desarrollaban un único mapeo para toda la población. La introducción de mapeos que no son uno-a-uno permite la posibilidad de capturar esquemas útiles, como se detalla más abajo. Ésta es quizá la mayor ventaja de este método.

Captura de esquemas

Supongamos un esquema de orden n que consiste en r bits a 0 y s bits a 1 (entonces n = r + s). Los bits a 0 forman juntos un subesquema de orden r llamado subesquema-0, mientras que los bits a 1 forman un subesquema de orden s llamado subesquema-1. Por ejemplo, el esquema 1*0*1*0*1 de orden 5 consiste en un subesquema-1 1***1***1 de orden 3, y un subesquema **0***0** de orden 2. Según los bits definidos de un subesquema emergen durante la evolución, el sistema intentará mantener los mismos valores de bits en dichas posiciones.

Debido a que se permite que los cromosomas de mapeo evolucionen libremente para formar un mapeo no uno-a-uno, la presión de selección podría mapear la posición de un bit definido en la posición de otro bit definido en el mismo subesquema. El resultado es una formación explícita de un esquema. Por ejemplo, el subesquema fenotípico 1***1***1 puede ser reconstruido también desde le patrón genotípico 1***0***0 si el cromosoma de mapeo es (0,1,2,3,0,5,6,0).

Una vez que un subesquema está formado explícitamente, el cambio de un bit genotípico puede disparar el cambio de múltiples bits definidos en el fenotipo. Así, la habilidad de alterar los valores de múltiples bits acorta mucho la distancia de Hamming entre dos hiperplanos distantes en el espacio de soluciones. Por ejemplo, la distancia entre 1***1***1 y 0***0***0 puede ser reducida a uno. Por tanto, la búsqueda puede escapar de un atractor defectivo o dirigirse hacia un óptimo global más rápidamente. En los problemas defectivos, a menudo el óptimo global es el complemento del atractor defectivo. En consecuencia, la formación explícita de subesquemas-1 o subesquemas-0 puede ayudar a que la búsqueda salte desde un atractor defectivo directamente hasta el óptimo global.

La teoría de esquemas propone, como hemos visto, que esquemas de orden superior y de gran longitud tienen menos posibilidades de sobrevivir porque son más propensos a ser destruidos por los operadores genéticos [3]. Por el contrario, en el método propuesto por Chow un orden superior de subesquema permite una reducción mayor de la distancia Hamming entre dos hiperplanos distantes.

Conclusiones

Los experimentos descritos en [19] demuestran que el algoritmo descrito en este apartado es muy efectivo a la hora de resolver algunos de los problemas defectivos. Más generalmente, resulta también eficiente en la resolución de otros problemas GA-Hard descritos en [14] y [15].

Nosotros hemos implementado un prototipo que aplica este mapeo evolutivo para solucionar el problema defectivo. Para ello hemos añadido un segundo cromosoma (el de mapeo) a cada individuo y modificado los operadores genéticos del modo descrito. Los detalles concretos pueden verse en el apartado 4.5 del capítulo dedicado a la implementación.

3 La tarjeta gráfica

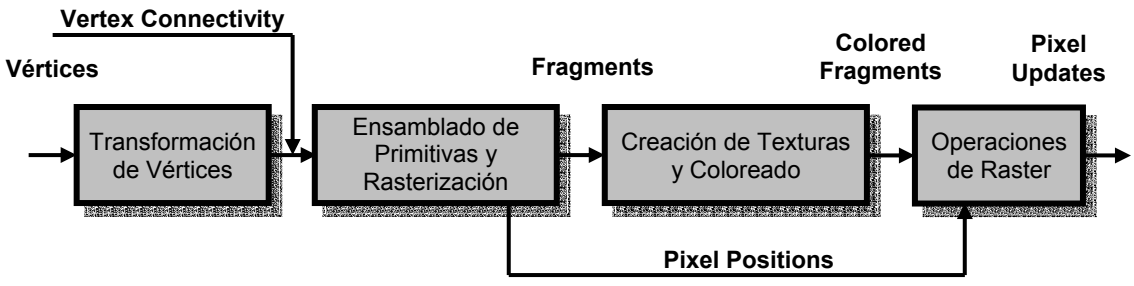
El presente trabajo surge como una forma de aprovechar la gran potencia de cálculo que nos ofrecen las tarjetas gráficas de los sistemas actuales, y que queda desaprovechada durante la ejecución habitual de un AG. La posibilidad de que el procesador de la GPU (*Graphics Processing Unit*) ejecute parte del AG, mejorando así el rendimiento global del sistema en la ejecución del mismo, se debe a la capacidad de ser programable que el hardware gráfico ofrece desde hace pocos años. Actualmente las tarjetas gráficas programables (como las nvidia GeForce3) vienen incorporadas en computadores comunes, que cualquier persona puede utilizar en casa o en el trabajo.

El uso eficiente de esta capacidad de cálculo inutilizada, en aplicaciones que no manejan gráficos, resulta una fuente de mejora del rendimiento disponible y relativamente fácil de encauzar. Para poder hacerlo hay que manejar una serie de modelos de lo que es y de cómo funciona una GPU.

Mientras que la CPU monoprocesador ejecuta los programas una sola vez, la GPU maneja otro modelo de programación en el que los programas se ejecutan una vez por cada elemento de datos (vértices o fragmentos). Este rasgo de paralelismo en la ejecución hace especialmente indicada la utilización de la tarjeta para la ejecución de AGs, también altamente paralelos como hemos visto.

3.1 La GPU como procesador de flujos

Para nuestros propósitos, la plataforma objetivo puede describirse como una arquitectura de procesamiento de flujos. Las tarjetas gráficas modernas utilizan una arquitectura de procesamiento de flujos segmentada para realizar una parte significativa de los cómputos en el proceso de renderizado:



Flujo lógico del proceso de renderizado

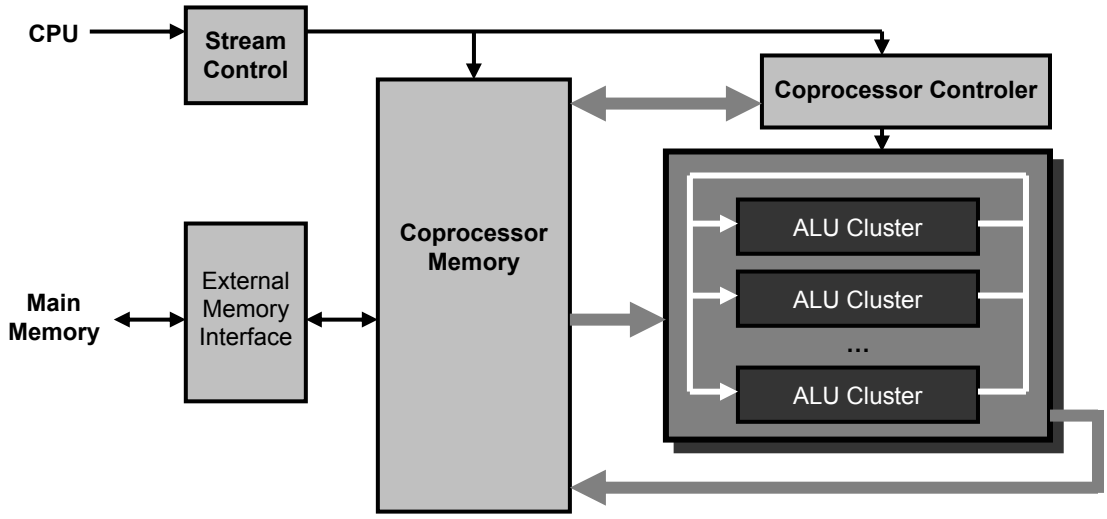
Un programa envía un conjunto de polígonos a la tarjeta gráfica. Cada vértice en un polígono es procesado de manera independiente atendiendo al *view point*. Una vez transformados, los vértices se ensamblan en sus polígonos constituyentes que son rasterizados convirtiéndolos en un conjunto de fragmentos. Tras algunas operaciones de *raster* los fragmentos finalmente seleccionados son enviados al *frame buffer* constituyendo los *píxels* que forman la imagen visualizada.

Dos de las etapas en el hardware gráfico moderno son programables, el procesador de vértices y el procesador de fragmentos (vertex y fragment engines/shaders). El primero se utiliza para realizar transformaciones sobre los atributos de cada vértice (normal, posición, color, textura, ...). El segundo en cambio se utiliza para realizar transformaciones sobre los distintos fragmentos que constituyen un polígono. Ambos procesadores son extremadamente paralelos, procesando varios elementos en paralelo y haciendo uso intensivo de unidades SIMD.

Este hardware programable de propósito específico puede ser utilizado en cómputos de propósito general aprovechando así sus cualidades paralelas. El principal obstáculo es que el algoritmo debe ser expresado como un programa de procesado de flujos (*streaming application*) adecuado para una de las dos unidades programables de la tarjeta.

Un algoritmo genético trabaja básicamente sobre un conjunto de individuos que pueden ser modelados como vectores. Los individuos forman entonces una población que puede ser modelada como un array 2D, en el que cada fila representa un individuo. El procesador de fragmentos está diseñado para trabajar sobre estructuras 2D con lo que parece la unidad más adecuada sobre la que mapear nuestro algoritmo genético.

A continuación mostramos una representación lógica del procesador de fragmentos:



Vista lógica de un procesador de fragmentos moderno.

El algoritmo debe ser representado como un conjunto de *kernels* que trabajan sobre algún flujo de datos de entrada produciendo uno o más flujos de datos de salida. En el contexto del procesador de fragmentos los *kernels* se denominan *fragment programs*.

Los flujos de datos de entrada iniciales son enviados a la GPU como texturas donde son almacenados en *buffers* de textura. El flujo de salida es escrito por la GPU en el *frame buffer*. Con el fin de poder utilizar estos resultados como entrada para otro *fragment program* con la mínima sobrecarga posible, es conveniente componer todas las texturas de entrada en un sólo OpenGl *pbuffer* y utilizar el modo *render to texture*.

Inicialmente los kernels y las texturas de entrada son transferidas de la memoria principal del sistema a la memoria integrada en la tarjeta gráfica, que puede operar en paralelo con la CPU. En la implementación final, sólo hay un reducido número de puntos de sincronización en los que la CPU dictamina el orden de ejecución de los distintos *kernels*.

3.2 El lenguaje Cg

Como vimos en la introducción, las GPUs actuales evolucionan muy rápidamente. Típicamente, una generación de producto dura sólo seis meses, y con cada nueva generación viene un incremento de dos dobleces en rendimiento. El rendimiento de los procesadores gráficos se incrementa en una proporción mucho mayor que la que establece la ley de Moore para microprocesadores [83]. Además de este incremento de rendimiento, cada año trae nuevas características hardware, soportadas por nuevas APIs.

Para los desarrolladores es muy difícil adaptarse a esta velocidad vertiginosa de progreso, pero por otro lado tanto los desarrolladores como los usuarios demandan más calidad de rendering e imágenes más realistas. Los usuarios no se preocupan por los detalles: simplemente desean que los juegos y otras aplicaciones interactivas tengan apariencia de película, con efectos especiales y animaciones. Los desarrolladores desean siempre más potencia, junto con mayor flexibilidad en el control de la capacidad de la GPU.

Según las APIs y las tecnologías subyacentes progresan, los desarrolladores luchan para adaptarse al cambio, y conseguir un compromiso entre los recursos hardware/software. Se necesita elevar el nivel de abstracción para interactuar con las GPUs. Las continuas actualizaciones y mejoras en el hardware y APIs son demasiado problemáticas si los desarrolladores están muy *cerca del metal*. Este problema se agudizó con la llegada de la programabilidad de las GPUs. La potente posibilidad de escribir *vertex* y *fragment programs* para ejecutarlos en la GPU requiere gran destreza, y sólo es factible para programas cortos.

Cuando el hardware GPU crece hasta permitir programas de miles de instrucciones, el ensamblado de código no es práctico. En vez de programar cada estado de *rendering*, cada bit, byte y palabra de datos o control a través de un lenguaje ensamblador de bajo nivel, queremos expresar nuestras ideas de una forma más directa, usando un lenguaje de alto nivel.

Esta necesidad justifica Cg (C para Gráficos). Cg es un lenguaje de alto nivel para programación gráfica basado en C, pero con modificaciones que facilitan la escritura de programas que compilan a código GPU altamente optimizado. Igual que C se derivó para exponer las capacidades específicas de los procesadores y permitir una abstracción

a más alto nivel, Cg permite la misma abstracción para GPUs, ocultando a los programadores los detalles de la implementación hardware. Puesto que Cg puede compilarse en tiempo de ejecución en cualquier plataforma, sistema operativo, y para cualquier hardware gráfico, los programas Cg son completamente portables. Además, los programas pueden adaptarse para funcionar correctamente en productos futuros: el compilador puede optimizar directamente para una nueva GPU destino que quizá no existía aún cuando el programa Cg original fue escrito.

El uso de un lenguaje de programación gráfica de alto nivel, en lugar de los lenguajes de bajo nivel del pasado, proporciona varias ventajas:

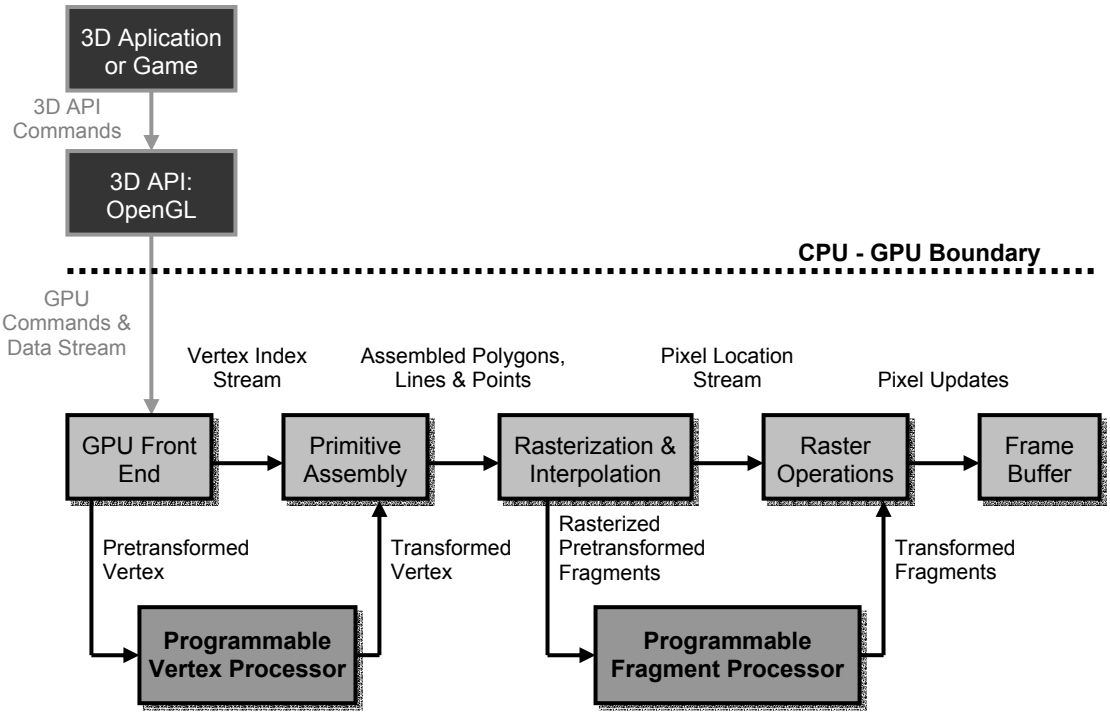
- Un lenguaje de alto nivel acelera el ciclo *tweak-and-run* cuando se desarrolla un *shader*. La posibilidad de prototipar y modificar rápidamente un *shader* es crucial para el desarrollo rápido de efectos de alta calidad.
- El compilador optimiza código automáticamente y realiza tareas de bajo nivel, tales como asignación de registros, que resultan tediosas y son fuente de error.
- El código de *shading* escrito en lenguaje de alto nivel es mucho más legible y fácil de entender, lo que permite crear fácilmente nuevos *shaders* simplemente modificando *shaders* escritos previamente.
- Los *shaders* escritos en lenguaje de alto nivel son portables a un mayor rango de plataformas hardware, frente a los escritos en código ensamblador.

La ventaja fundamental que Cg nos ha aportado personalmente a los desarrolladores del presente proyecto es la abstracción de una tecnología hardware desconocida para nosotros, permitiéndonos así trabajar en un nivel de abstracción más intuitivo, y obviar el aprendizaje de tecnologías complejas que hubieran dilatado considerablemente el desarrollo del trabajo.

3.2.1 Modelo de programación de Cg para GPUs

Las principales desigualdades existentes entre Cg y C provienen de la diferencia de modelos de programación para GPUs y para CPUs.

Las CPUs normalmente cuentan con un único procesador programable. En contraste, las GPUs tienen al menos dos procesadores programables, además de otras unidades hardware no programables. Los procesadores, las partes no programables del hardware gráfico, y la aplicación se conectan mediante flujos de datos. En el siguiente esquema mostramos el modelo Cg de la GPU:



Modelo Cg de la GPU

El lenguaje Cg permite escribir programas para ambos procesadores: el *vertex processor* y el *fragment processor*. Estos programas reciben los nombres de *vertex programs* y *fragments programs* (también *pixel programs* o *pixel shaders*), respectivamente.

En el presente trabajo, todos los programas Cg desarrollados para la implementación de AGs sobre GPU son *fragments programs*. Un *fragment program* se ejecuta repetidamente en la GPU una vez por cada elemento de datos en un flujo, en este caso una vez para cada fragmento.

3.3 Especificaciones técnicas de la GPU

La tarjeta gráfica instalada en el sistema donde se han realizado las ejecuciones y tomado las medidas de rendimiento de los AGs implementados es el modelo nvidia GeForce FX 5950 Ultra, cuyas especificaciones incluimos a continuación:

** Maximum in a single rendering pass*

Specifications/Performance	
Graphics Core	256-bit
Memory Interface*	128-bit
Memory Bandwidth*	30.4GB/sec.
Fill Rate	3.8 billion texels/sec.
Vertices per Second	356 million
Memory Data Rate	950 MHz
Pixels per Clock (peak)	8
Textures per Pixel**	16*
Dual RAMDACs	400 MHz

Key Features	
CineFX 2.0 Engine	The second-generation CineFX 2.0 engine powers advanced pixel and vertex shader processing and true 128-bit color precision. Delivering double the floating-point pixel shader power of the previous CineFX engine, CineFX 2.0 produces a visible performance boost through its more efficient execution of pixel shader programs.
UltraShadow Technology	Powers the next-generation of complex, realistic shadow effects by accelerating shadow generation. Accurate shadows that effectively mimic reality without bogging down frame rates are one of the keys to more believable game environments.
Intellisample HCT	Second-generation Intellisample technology delivers up to a 50-percent increase in compression efficiency for compressing color, texture, and z-data, and powers unprecedented visual quality for resolutions up to 1600 x 1280.

4 Implementación y Desarrollo

Detalles técnicos

El desarrollo del presente proyecto ha sido realizado en Visual C++.net por razones de compatibilidad con el compilador de Cg.

S.O.: Windows XP Service Pack 2.

Tarjeta gráfica (GPU): nvidia GeForce FX 5950 Ultra.

Procesador (CPU): Pentium IV, 3GHz, 512 MB (RAM).

En el proceso de desarrollo del proyecto se pueden distinguir tres etapas:

1) Implementación de AGs sobre la CPU.
2) Implementación de AGs sobre la tarjeta gráfica.
3) Implementación de AGs paralelos, con un hilo sobre la CPU y otro sobre la tarjeta gráfica.

Las mediciones han servido de guía durante el proceso completo, indicándonos nuevos caminos a explorar y posibles mejoras de las diferentes implementaciones. Además, los objetivos iniciales han ido evolucionando a medida que los estudios realizados nos abrían nuevas posibilidades de aprovechamiento de la tarjeta gráfica. Así por ejemplo, frente a la idea inicial de realizar la migración entre las poblaciones de sendos hilos de modo síncrono, de modo que ambos procesadores realizasen el mismo número de generaciones, se abrió la posibilidad de migrar asíncronamente para aprovechar aún más la capacidad de cómputo de la tarjeta y dejar que ésta hiciese tantas generaciones como le fuera posible. En la implementación final la CPU es la que marca los eventos de migración (cuando se completan las generaciones especificadas por el usuario), y hasta entonces la tarjeta se mantiene activa, de modo que no se produce desaprovechamiento de ésta.

De igual modo, las especificaciones iniciales se han revisado y corregido ante las restricciones hardware y software encontradas durante el proceso de desarrollo e implementación. La dificultad más destacable en este sentido la hemos encontrado al paralelizar el AG. Dado el papel de ayuda que desempeña la tarjeta gráfica en nuestro planteamiento, la idea inicial fue ejecutar la implementación del AG correspondiente a la CPU en el hilo principal y la implementación sobre la tarjeta en un hilo secundario. Sin embargo, las funciones provistas por OpenGL plantean serias dificultades a la hora de ejecutarse sobre threads.

Para salvar este contratiempo tuvimos que invertir el modelo original y ejecutar la implementación para la tarjeta sobre el proceso principal (sin hilos) de la aplicación, y lanzar un hilo con el algoritmo para CPU. Los resultados obtenidos no se ven alterados por este cambio, y los objetivos alcanzados siguen coincidiendo con la tesis inicial de este proyecto.

Como ya hemos visto, el AG consiste fundamentalmente en un bucle de evolución que se aplica, sobre una población, un número de veces prefijado por el usuario (número de generaciones). El esquema fundamental implementado tanto en la CPU como en la GPU es el ya conocido:

```
generación de la población inicial
evaluación de la población inicial
while no se cumpla la condición de parada do
   selección
   cruce
   mutación
   evaluación de los individuos
end while
```

En los apartados siguientes veremos las diferentes formas de implementar los diferentes pasos de este proceso según la ejecución sea sobre la CPU o sobre la tarjeta gráfica, así como las diferentes estructuras de datos usadas para mantener la información involucrada en los AGs.

Para estudiar los potenciales del uso de la tarjeta gráfica como ayuda a la CPU en la ejecución de AGs, hemos utilizado problemas típicos del campo de la programación evolutiva. Estos problemas son especialmente indicados para nuestros propósitos por varias razones: son sencillos de implementar, se conoce la solución a priori y por tanto resulta trivial comprobar el buen funcionamiento de la implementación, y existe abundante documentación (y mediciones) sobre ellos a modo de referencia. Los problemas implementados pretenden representar a los principales tipos, o al menos a los de mayor interés, para dar así una idea amplia de las posibilidades de nuestra propuesta.

Se han implementado los AGs correspondientes a tres problemas, definidos por sus correspondientes funciones de aptitud (fitness):

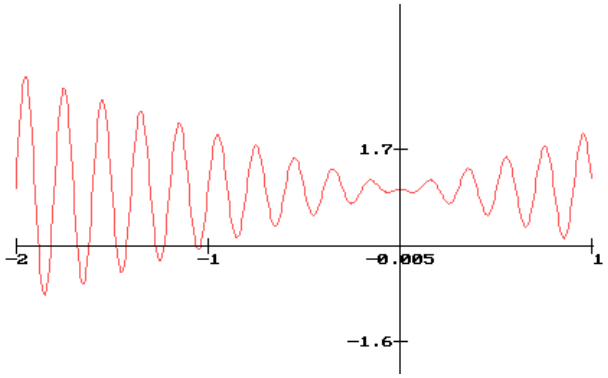
- **Onemax**: resuelve el problema de obtener la cadena binaria con mayor número de 1s posible. Para ello define como función de aptitud:

$$F_{\text{onemax}}(\bar{x}) = \sum_{i=1}^{n} x_i \ ,$$

  siendo $\bar{x}$ el vector de genes (cromosoma), n la longitud del cromosoma y x_i el gen i-ésimo (con valor 0 ó 1).

- **Funciones Multimodales:** buscan un óptimo (máximo ó mínimo) global en una función con muchos óptimos locales, aunque sin llegar a ser funciones defectivas. Sus aptitudes vienen descritas típicamente por funciones trigonométricas. Nosotros hemos escogido la siguiente:

$$F_{math}(x) = -x \cdot sen(10\pi x) + 1, \text{ para x en } [-2,1].$$



- **Funciones trampa** (o defectivas): buscan un óptimo global en una función de fitness defectiva, que les conduce a óptimos locales. La función defectiva escogida ha sido:

$$F_{ftrap}(\bar{x}) = \begin{cases} k - F_{one\max}(\bar{x}) - 1, & si F_{one\max}(\bar{x}) < k \\ k & si F_{one\max}(\bar{x}) == k \end{cases}, \text{ para k = 3 ó 4}$$

Además se ha realizado una implementación alternativa que soluciona el problema defectivo, obteniendo un fitness considerablemente mejor al habitual. Esta mejora se ha desarrollado a partir del algoritmo de mapeo evolutivo planteado en el capítulo de Algoritmos Genéticos, y se explica en el último apartado del capítulo, puesto que tanto los operadores como las estructuras de datos usados no son comunes con el resto de implementaciones.

4.1 Implementación sobre la CPU

Puesto que la ejecución de estas versiones de los AGs se realiza sobre la CPU, la implementación es la natural. Por tanto, en este apartado no entraremos en detalles que no presentan interés para nuestros objetivos. Simplemente plantearemos grosso modo las estructuras de datos empleadas y la forma en que los diferentes operadores genéticos actúan sobre ellas, sobre todo para establecer un punto de comparación con la implementación sobre tarjeta.

Los números aleatorios, parte esencial de los AGs, se van generando en esta implementación a medida que se van necesitando, invocando para ello a la función rand(). Este punto, que aquí resulta trivial, resultará de extrema importancia en la implementación sobre la tarjeta, debido a que ésta no es capaz de generar números aleatorios.

4.1.1 Estructuras de datos

La población se ha implementado como un vector de referencias a individuos:

```
typedef vector<Individuo*> TPoblacion;
```

Cada individuo consiste, además de en una serie de parámetros de AG, en una cadena binaria (cromosoma de genes) que hemos implementado mediante un vector de enteros (con elementos 0 ó 1):

```
typedef vector<int> TGenes;
```

4.1.2 Operadores evolutivos

Como hemos visto, hay diferentes posibilidades para los distintos operadores genéticos. La elección suele realizarse en función del problema a resolver y de los objetivos concretos de cada aplicación. En nuestro caso, las elecciones han venido determinadas fundamentalmente por las restricciones de la tarjeta gráfica. Puesto que, a fin de que las ejecuciones de los AG sobre la tarjeta y sobre la CPU sean comparables, hemos implementado los mismos operadores genéticos, se han escogido las variantes más adecuadas a la estructura y características de la tarjeta para todas las implementaciones.

Operador de Selección

La selección de individuos se realiza por torneo. Para ello se utiliza una estructura de datos auxiliar que se corresponde a una población intermedia, en la que vamos copiando los individuos que se seleccionan para la siguiente generación. Basta con construir una lista con los índices (en la población de partida) de los individuos que se van seleccionando (`int[tam_pob]`).

Para implementar elitismo (salvamos siempre el mejor individuo) hemos de pasar siempre el individuo de mejor fitness a la siguiente generación. Para ello simplemente establecemos como primer elemento del array de índices de individuos seleccionados el índice del individuo de mayor fitness (obsérvese que, en el bucle de evolución, antes de la selección se invoca a la evaluación de la población). De este modo el primer individuo de la población será siempre el mejor, al menos de la generación anterior, lo cual facilitará la operación de migración como veremos más adelante.

Por razones de eficiciencia a la hora de aplicar el operador de cruce, nos resulta más conveniente duplicar el mejor individuo de modo que aparezca en las dos primeras posiciones de la nueva población. Para el resto de posiciones (2..tam_pob-1) se realiza la selección por torneo.

Una vez construido el array de índices de individuos seleccionados, se construye la población para la siguiente generación copiando los individuos seleccionados de la población de partida según indique el array de índices.

Operador de Cruce

El operador cruce implementado es el que intercambia (si se realiza el cruce) los cromosomas de los progenitores por un único punto elegido al azar. El modo de escoger a los padres, sin embargo, no es el habitual.

Por razones concernientes a las características de la tarjeta, la reproducción se simplifica considerablemente si se sigue un orden secuencial en la selección de los padres, en lugar

de elegirlos al azar para cada cruce. Así, se ha implementado una reproducción en la que el primer individuo se cruza (si procede) con el segundo, produciéndose dos nuevos individuos, el tercero con el cuarto, el cuarto con el quinto, etc. Puesto que en el paso previo de selección se introduce desorden en la población, el determinismo introducido en este paso no afecta al buen funcionamiento del AG (a la vista de los resultados obtenidos).

Nótese que, puesto que el primer individuo y el segundo corresponden en realidad al mismo (élite de la población), la aplicación del cruce pasará automáticamente el mejor individuo a la siguiente población: En caso de que se produzca el cruce, puesto que los dos padres tienen cromosomas idénticos, los hijos generados serán iguales; si no se realiza el cruce pasarán directamente los padres, obteniéndose en ambos casos el efecto deseado.

Operador de mutación

El operador de mutación escogido puede mutar cada gen de cada individuo. Su implementación es trivial, recorriéndose el cromosoma de cada individuo y decidiendo (en función de la probabilidad de cruce y los números aleatorios obtenidos) si se muta cada gen o no.

Los dos primeros individuos de la población (élite) no se exponen a este operador destructivo.

Evaluación

En el paso de evaluación se recorre secuencialmente la población calculando el fitness de cada individuo, dado por la función de aptitud. Como hemos visto, esta función viene dada por el problema concreto que resuelve el AG.

El cálculo de la aptitud para los problemas onemax y defectivo se reduce a aplicar las correspondientes funciones descritas anteriormente. Sin embargo, la función trigonométrica requiere un paso previo: la decodificación.

Puesto que en este problema concreto la función de aptitud consiste en una función matemática continua, se define para una variable x en R. Sin embargo, esta función debe valorar cadenas binarias (individuos). Para poder establecer una correlación entre ambos dominios discretizamos el intervalo $[x_{min}, x_{max}]$ en una cantidad de puntos 2^{lcrom}, de modo que cada posible individuo (para la longitud de cromosoma establecida) se corresponde con un punto del dominio de la variable x. Por tanto, antes de aplicar la función de aptitud (definida para la variable real x) calculamos a qué punto del dominio de dicha función corresponde el individuo a evaluar:

$$x(\bar{x}) = x_{\min} + dec(\bar{x}) \cdot \frac{x_{\max} - x_{\min}}{2^{lcrom} - 1},$$

donde x es la variable real para la que se define la función trigonométrica de aptitud, x_{min} y x_{max} son los valores extremos que puede tomar dicha variable en R, $\bar{x}$ es el vector de genes del individuo y dec($\bar{x}$) es el valor decimal de decodificación de la cadena binaria del cromosoma.

4.2 Implementación sobre la GPU

A la hora de transformar un algoritmo secuencial en un programa de procesado de flujos debemos tomar una serie de decisiones de diseño. Consideramos dos problemas principales:

1) El algoritmo debe ser descompuesto en una serie de *kernels* sencillos que trabajan sobre flujos de datos homogéneos produciendo nuevos flujos de datos.
2) Una vez realizada esta división, los datos deben ser reorganizados en los flujos necesarios para los distintos *kernels*. Estos flujos de datos se agrupan en una sola textura 2D.

En el procesamiento de kernels estamos limitados por las características de la plataforma objetivo. Por ejemplo, ya hemos visto que no tenemos la posibilidad de generar números aleatorios en la tarjeta gráfica. En nuestro caso, solucionamos el problema introduciendo un kernel que obtiene nuevos números aleatorios a partir de un conjunto de ellos presentes en la textura. Estos números aleatorios iniciales son generados en la CPU y se escriben en la textura inicial. El flujo de kernels es controlado por la CPU, es decir, la GPU actúa como un coprocesador que inicia la ejecución de un determinado kernel sobre un determinado flujo de datos de entrada cuando se lo ordena la CPU.

Cuando hablamos de algoritmo genético sobre la GPU, nos referimos a un algoritmo genético en el que la CPU lleva el control del mismo. Es decir, tanto el bucle de ejecución principal como la definición de los flujos de datos antes mencionados, se llevan a cabo en la CPU y, por tanto, el algoritmo consume recursos del procesador. Sin embargo, la ejecución de los operadores genéticos se realiza en la GPU y no consumen tales recursos. Además, la población del algoritmo está almacenada en la memoria de la tarjeta, con lo que tampoco consume recursos de la memoria principal. Aquí reside realmente la ayuda que proporciona la GPU en la mejora de rendimiento en la ejecución de algoritmos genéticos.

El siguiente algoritmo representa el flujo lógico de *kernels* en los que hemos descompuesto el AG.

```
Initialize_texture();

foreach gene in an individual
   K1: Evaluate_gen_column();
end foreach;

foreach generation
   K3: Select_individuals();
   K2: Refresh_random_stream();
   K4: Cross_Selected_Individuals();
   K2: Refresh_random_stream();
   foreach gene in an individual
      K5: Mutate_gen_column();
      K2: Refresh_random_stream();
   end foreach;
   foreach gene in an individual
      K1: Evaluate_gen_column();
   end foreach;
end foreach;
```

El proceso comienza por la transmisión de la textura inicial de la CPU a la GPU. Esta textura está lógicamente dividida en 5 regiones. El *frame buffer* es mapeado sobre alguna región de esta textura gracias al uso del modo *render to texture*. Hasta seis regiones de la textura puede tomar cada *kernel* como flujos de datos de entrada.

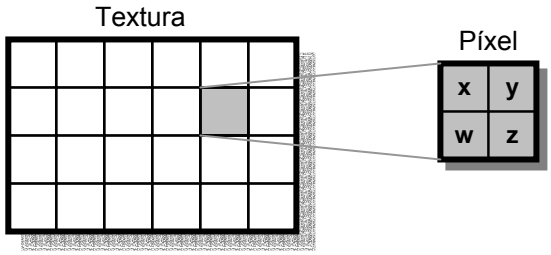
4.2.1 Estructuras de datos

Para la implementación sobre la tarjeta gráfica, debemos mantener todos los datos involucrados en los AGs en la textura de GPU. Por tanto, el primer paso consiste en ver qué datos necesitamos realmente y cómo organizarlos eficientemente sobre la textura.

Los datos que se necesitan son:

- La población, que consiste en una lista de individuos, cada uno de ellos a su vez es una secuencia binaria de genes.
- Las probabilidades que maneja el AG a modo de parámetros: probabilidad de cruce y probabilidad de mutación.
- Fitness asociado a cada individuo, dado por la función de aptitud de cada problema concreto.
- Números aleatorios para realizar las diferentes etapas del proceso evolutivo.

Por otro lado, la textura está formada por una secuencia de píxeles, siendo la interpretación habitual la de una matriz bidimensional. Cada píxel puede almacenar cuatro datos tipo *float*, siendo éstos accedidos individualmente mediante las correspondientes coordenadas:



Teniendo en cuenta tanto las necesidades del algoritmo como las restricciones y características de la textura GPU, se plantea la zona de textura utilizada a modo de matriz de píxeles donde cada fila se corresponde a un individuo de la población. Por tanto, contamos con una textura rectangular de altura igual al número de individuos (tamaño de la población), y de anchura determinada por la información que debe mantenerse de cada individuo.

El dato fundamental que debe mantener un individuo es su cromosoma que, para los problemas afrontados en el presente trabajo, consiste en una cadena binaria. Por tanto, la implementación natural es emplear una componente de píxel (float) para almacenar un gen (entero en [0,1]). En consecuencia, en cada píxel de la zona destinada a los cromosomas se almacenan cuatro genes, y en consecuencia el ancho de dicha zona será de M/4 (siendo M la longitud de cromosoma).

Al igual que ocurría en las implementaciones sobre la CPU, algunas operaciones (como la selección o la reproducción) necesitan construir poblaciones intermedias a fin de no destruir los datos originales. La forma de hacerlo sobre la textura GPU es doblando la zona de genes, de modo que en cada momento una de estas áreas actúe como zona de lectura (la que contiene los datos válidos) y la otra como zona de escritura (la que recibe los resultados, que pasan a ser los nuevos datos válidos). A lo largo del algoritmo estas zonas se van intercambiando de modo eficiente.

El otro dato asociado a cada individuo es su valor de fitness, que también habrá que almacenar en la fila correspondiente. Para almacenar este valor basta una componente de píxel, de modo que tendremos un píxel con tres componentes inutilizadas (más adelante veremos cómo se han utilizado). Puesto que este valor depende únicamente de los genes actuales (y no del valor de fitness anterior), basta con llevar una única copia.

Por otro lado, para ejecutar el AG se necesitan otros datos no asociados a los individuos concretos, pero que aún así hay que mantener: los números aleatorios y los valores de probabilidades. A continuación veremos cómo hemos afrontado el problema de optimizar su almacenamiento y manejo.

Números aleatorios

Los números aleatorios se utilizan para realizar ciertas operaciones sobre los individuos de la población:

- **selección**: necesita tres números aleatorios para calcular cada individuo seleccionado para la nueva población: dos para producir las posiciones de los dos individuos a enfrentar en torneo, y uno para seleccionar al individuo ganador (junto con sendos valores de fitness)
- **cruce**: puesto que la elección de los padres es determinista, sólo necesita dos números aleatorios: uno para determinar el punto de cruce y otro para decidir si se realiza el cruce (junto con la probabilidad de cruce)
- **mutación**: dado que permitimos la mutación de cada gen de cada individuo, necesitamos un píxel de aleatorios para cada píxel de cada individuo (junto con la probabilidad de mutación)

A la vista de las necesidades planteadas parece que, aunque se trate de datos externos a la población, podemos asociar los números aleatorios a los individuos. De este modo podemos aprovechar mejor el paralelismo y el modo en que la GPU mapea las zonas de lectura sobre las de escritura, si cada individuo accede a sus números aleatorios asociados. Este planteamiento nos lleva a situar los números aleatorios a modo de columnas.

Para disponer de los números aleatorios necesarios durante el proceso evolutivo se parte de una población inicial de números generados en la CPU. Cuando, tras una operación del AG, se agotan los números aleatorios, se ejecuta el *kernel* K2 para regenerar el juego de números aleatorios manteniendo las propiedades estadísticas de los mismos.

Para recombinar los números aleatorios necesitamos duplicar la zona de la textura correspondiente estableciendo en cada momento una zona como de lectura y la otra como zona de escritura (igual que hacíamos para la zona de genes). El kernel K2 se ejecuta en paralelo sobre la zona de escritura, de modo que para cada píxel de escritura se establece cada coordenada del siguiente modo: se lee la misma coordenada del píxel correspondiente a la zona de lectura de números aleatorios, y se multiplica por el número de filas de la textura menos uno. Puesto que cada número aleatorio es un real en [0,1], obtenemos un real en [0, tam_pob-1] que nos da una fila aleatoria de la textura (tomando para ello la parte entera de la cifra obtenida). Por último leemos el número aleatorio almacenado de dicha fila en la componente que estamos calculando. Realizando esta operación para cada componente de cada píxel obtenemos un nuevo juego de números aleatorios.

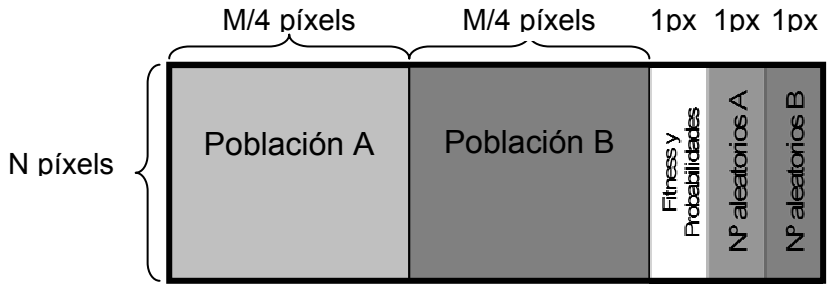
Este mecanismo resulta válido (salvaguarda las propiedades estadísticas de los números aleatorios) si se parte de una población inicial de números aleatorios lo suficientemente extensa, es decir, si el tamaño de la población (y por tanto la altura de la textura) es lo suficientemente grande.

Valores de probabilidad

Los parámetros del AG correspondientes a la probabilidad de cruce y a la probabilidad de mutación son globales y constantes para todo el algoritmo. Sin embargo, la necesidad de leer estos valores cada vez que se aplica el operador de cruce o el de mutación sobre la población, junto con el mecanismo de solapamiento de zonas de lectura y escritura que utiliza la tarjeta al ejecutar los correspondientes *fragment programs*, hacen que la replicación de estos datos para cada individuo simplifique el algoritmo.

Puesto que hemos de almacenar estos dos datos para cada individuo, y recordando que el píxel que almacena el fitness del individuo tiene tres coordenadas libres, finalmente se optó por emplear este único píxel para mantener estos tres datos de cada individuo.

A continuación se muestra la vista lógica de la textura final para un algoritmo genético con N individuos de M genes:



4.2.2 Operadores evolutivos

Los operadores evolutivos se implementan en la tarjeta gráfica a modo de kernels, como hemos visto. Para cada uno de ellos se definen las coordenadas de todas las zonas de lecturas necesarias para cada operación y las coordenadas de la zona de escritura, y se ejecuta el *fragment program* correspondiente de modo totalmente paralelo sobre cada

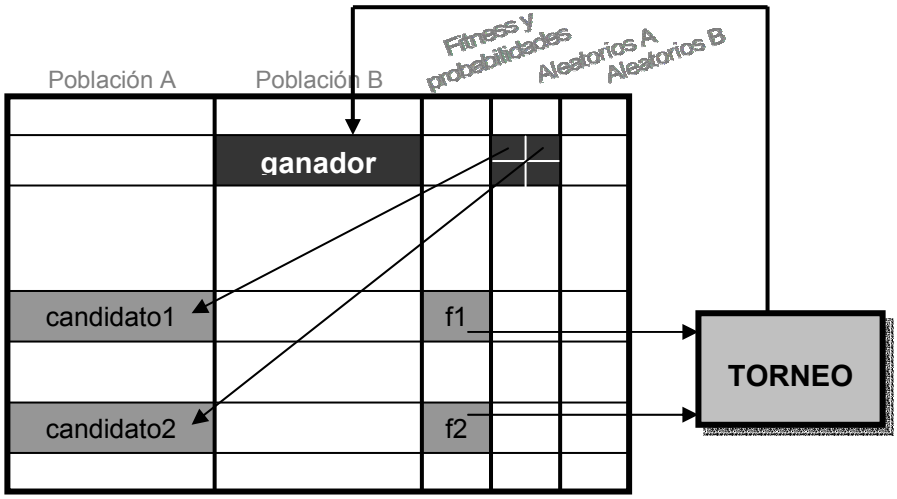
píxel constituyente de la zona de escritura. A este fenómeno se le llama mapeo del *fragment program* sobre la textura, tomando como *flujo de entrada* las diferentes zonas de lectura definidas.

Operador de Selección

Las características paralelas de la arquitectura objetivo imponen el uso de un método de selección altamente paralelo, nosotros optamos por el torneo.

Para implementar elitismo, antes de mandar a ejecutar el *kernel* K3 (correspondiente a la selección), ha de realizarse la promoción de la élite a la nueva población. Para ello ha de localizarse el mejor individuo de la Población A y, puesto que se trata de una operación de comparación de valores totalmente secuencial, este paso previo ha de ejecutarse en la CPU. Una vez localizado el mejor individuo, se ejecuta un *fragment program* de copia de píxeles, definiéndose como zona de lectura la correspondiente al cromosoma del mejor individuo (en la Población A), y como zona de escritura las dos primeras posiciones de la nueva población (Población B).

Para el resto de posiciones de la Población B (2 a tam_pob-1), el proceso de selección puede representarse esquemáticamente del siguiente modo:



La zona de escritura en este caso es la Población B, sobre la que debe mapearse el *frame buffer*. El kernel K3 recibe como flujos de entrada las áreas que contienen la aptitud de cada individuo y los parámetros de probabilidades del AG, los números aleatorios, y la Población A.

Para cada individuo de la Población B, se escogen dos individuos de la Población A (mediante los números aleatorios) y se hacen competir mediante torneo.

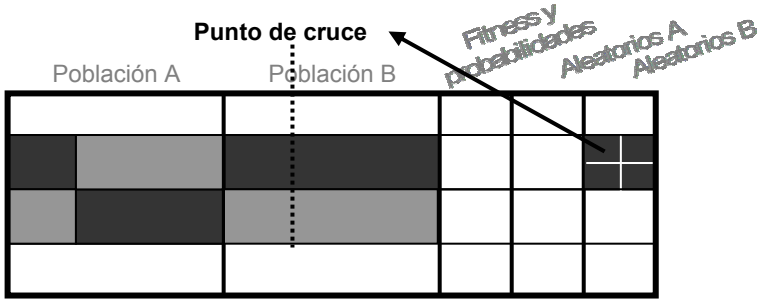
El individuo ganador es seleccionado y copiado en el *frame buffer*, en su lugar correspondiente de la Población B. Debemos remarcar que la selección de los N nuevos individuos se realiza en paralelo.

Tras la operación de selección, regeneramos el juego de números aleatorios invocando al kernel K2.

Operador de Cruce

El kernel K4 se computa a continuación, realizando el cruce de los individuos seleccionados. La zona de escritura, sobre la que se mapea el *frame buffer,* corresponde al área marcada como Población A. Como entrada se toman los flujos de datos que contienen los individuos previamente seleccionados (Población B), y los conjuntos de números aleatorios y probabilidades. K4 cruza los individuos en una posición par dentro de la población con los de una impar.

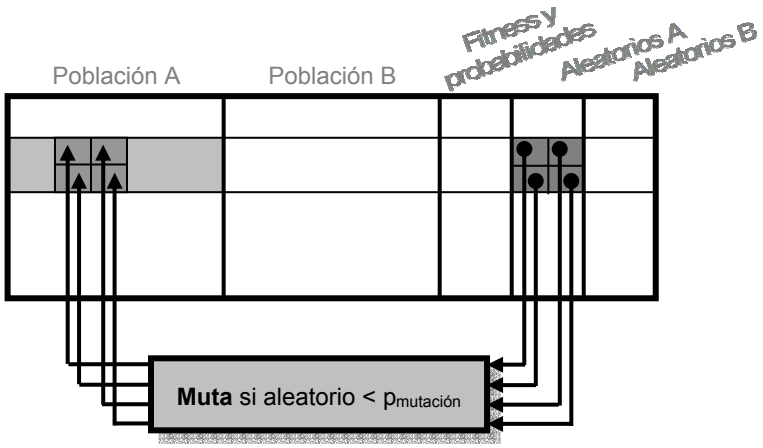
Se selecciona para los dos individuos un punto de cruce común aleatorio, utilizando para ello el flujo de píxeles aleatorios. Se decide si se cruzan usando la probabilidad de cruce y otro aleatorio. Los dos nuevos individuos se copian al *frame buffer;* si se realiza el cruce, cada uno de los hijos tiene una parte de cada padre; si no se cruzan, se copian los individuos originales. En la siguiente figura se muestra el caso en que se realiza el cruce:



Operador de Mutación

Finalmente, los nuevos individuos se mutan (K5). En este caso el *frame buffer* se mapea sobre cada columna del área marcada como Población A, que se utiliza también como flujo de datos de entrada. Esto es posible porque sólo se requiere los datos del píxel a mutar, es decir, no necesitamos los datos originales del individuo.

Tras la ejecución del kernel K5 sobre una columna de la Población A hemos de recombinar los números aleatorios, debido a que el operador de mutación implementado necesita un píxel de aleatorios para cada píxel de genes.



Se decide si los genes contenidos en cada píxel de una columna en la Población A deben ser mutados en función de los números aleatorios encontrados en el flujo de entrada y de la probabilidad de cruce: cada coordenada del píxel de genes considerado muta según el valor de la misma coordenada del píxel de aleatorios.

Evaluación

El kernel de evaluación (K1) obtiene la aptitud de cada individuo y, puesto que el valor de ésta es diferente para cada una de las implementaciones realizadas, los *fragment programs* correspondientes son propios de cada problema.

Para los AG correspondientes a los problemas onemax y defectivo, las funciones de aptitud son funciones del número de 1s del cromosoma del individuo. La suma implicada de valores no permite un paralelismo por columnas (aunque sí por filas), de modo que, en este caso, el *frame buffer* es mapeado sobre la zona de aptitud de la textura. Como flujo de entrada toma una sola columna de la Población A, calculándose el fitness de cada individuo (gracias al paralelismo existente dentro de cada columna) acumulativamente según se recorren las columnas de la zona de textura donde se almacenan los genes.

Recordamos que en el caso de la función trigonométrica era necesario un paso previo al cálculo de la aptitud de un individuo: la traducción del genotipo del individuo a un punto del dominio de la función de fitness. Para obtener un buen rendimiento, hemos combinado ambos procesos en el mismo kernel de evaluación. Así, también en este caso el *frame buffer* es mapeado sobre la zona de aptitud de la textura tomando como flujo de entrada una columna de la Población A. El código del *fragment program* consiste esencialmente en una sentencia if: si nos hallamos en una columna no final (0 a lcrom-2) estamos calculando acumulativamente la decodificación binaria del cromosoma en la zona de aptitud; si estamos en la columna final (lcrom-1) la decodificación se ha completado y la zona de aptitud contiene el valor decimal del cromosoma. En el primer caso, añadimos al valor calculado hasta el momento la aportación de los cuatro genes del píxel actual. En el segundo caso, además, calculamos el punto del dominio correspondiente al valor decimal del cromosoma y, a continuación, aplicamos la función de aptitud para obtener el valor de fitness definitivo.

El pseudo código correspondiente a este *fragment program* es:

```
//Calculo el valor binario del cromosoma
float pot = pow(2, 4*columna_gen); //potencia de inicio del píxel
fitness += pot*gen.x; pot *= 2;
fitness += pot*gen.y; pot *= 2;
fitness += pot*gen.z; pot *= 2;
fitness += pot*gen.w; pot *= 2;

//Estamos en la última columna -> decodificación finalizada
if (columna_gen == lcrom - 1)
   //Calculo punto de la función correspondiente al individuo
   fitness = xmin + fitness*(xmax - xmin)/(pow(2,lcrom)-1);
   //Calculo valor de fitness según función de aptitud
   fitness = - fitness*sin(10*fitness*PI) + 1;
```

4.3 Programación de la GPU

Como se puede observar, el funcionamiento de la GPU es muy distinto al de un procesador corriente. Por tanto, programar sobre la GPU conlleva diferencias respecto a la programación normal de procesador a la que estamos habituados.

Los programas para la GPU se desarrollan en el lenguaje de implementación Cg. Este lenguaje, desarrollado por NVIDIA, proporciona el soporte para manejar las tarjetas gráficas programables. Para ello, se necesita un compilador de Cg, proporcionado por la propia NVIDIA, que será el encargado de compilar los programas en este lenguaje.

El compilador de Cg

Como ya dijimos, la elección de Visual C++ se debe a la facilidad que nos proporciona a la hora de conectar con el compilador Cg. Para poder desarrollar programas en los que la CPU interactúe con la tarjeta, debemos conectar el compilador de Cg a nuestro Visual C++, además de proporcionar las rutas de las librerías OpenGL necesarias para el programa que estemos desarrollando.

Esto se consigue editando las Opciones de Visual C++, bajo la pestaña de Herramientas. Aquí debemos indicar la ruta del compilador Cg en los directorios de ejecutables que reconoce Visual, y las librerías que necesitamos (estáticas o dinámicas) en los directorios de librerías (estáticas o dinámicas).

Con esto, Visual C++ puede encontrar el compilador Cg para compilar los programas escritos en este lenguaje (.cg). Tendremos que indicarle ahora qué fichero queremos que compile. Para ello, cada fichero .cg tiene una hoja de propiedades. Dentro de esta hoja de propiedades podemos encontrar un paso de generación personalizada. Este paso es el que usaremos para compilarlos.

Visual C++ compila todo fichero del proyecto con su compilador, siempre que lo reconozca (por ejemplo, compila los ficheros C++ porque internamente tiene la orden de hacerlo). Sin embargo, no sabe qué hacer con los ficheros .cg, porque no conoce estos ficheros.

En el paso de generación personalizada podemos indicarle qué hacer con ellos, realizando los siguientes ajustes:

- en la Línea de Comandos:

  `"$(CG_BIN_PATH)\cgc" "$(InputPath)" -o "$(InputName)".fp -profile fp30`

  Esta línea indica a Visual C++ que debe usar el compilador "cgc", que se encuentra en la ruta de directorios de ejecutables antes introducida, para compilar los ficheros Cg siguiendo el perfil fp30.

- en Resultados: `$(InputName).fp`

  Esta línea hace que los programas objeto se denominen .fp.

Debido a que los procesadores CPU tienen casi las mismas capacidades, los compiladores de C soportan todas estas capacidades. Sin embargo, no se ha llegado en GPUs hasta este nivel de generalidad, por lo que el compilador de Cg incluye una gama de perfiles de compilación, en cada uno de los cuales se define un subconjunto del total del lenguaje Cg que es soportado para un determinado hardware.

En nuestro caso se usa el *OpenGL NV30 fragment programs*. Éste es un perfil avanzado que proporciona soporte para *fragment programs* complejos usando OpenGL [83].

Una vez hecho esto debemos asegurarnos de que, cada vez que modifiquemos el código de un programa Cg, generamos de nuevo todo el proyecto. Si no, Visual C++ no lanzará la generación personalizada, y no se modificará el archivo objeto del *fragment program*.

Uso de ficheros Cg

Una vez compilados, se obtienen los *fragment programs* correspondientes a los Cgs, y se pueden usar en el código. Para ello tenemos que crear objetos *CFragmentProgramNV*, y vincular el código objeto resultante de la compilación a cada uno de ellos. Cuando queramos ejecutar uno de ellos tendremos que activarlo (mediante un bind). Esto hace que el *fragment program* que se ejecute sobre la tarjeta sea aquel que actualmente esté vinculado, y realice el código que tenga asociado.

Básicamente, a la hora de usar la tarjeta, tenemos que pasar los datos de la población a ella. Estos datos serán guardados en una textura, que será la que maneje la GPU. Para ello, tenemos que comenzar creando una ventana GLUT, para poder usar las OpenGL. Además, debemos inicializar las librerías OpenGL que necesite nuestro programa (si es que tienen que ser inicializadas). Por ejemplo, en nuestro caso inicializamos la librería GLEW, que inicializa los puntos de entrada de esta extensión de OpenGL.

A partir de este punto podemos usar texturas. Lo primero será crearla, dándole el tamaño adecuado (con un objeto CGPU). Al crear la textura, se vincula, para poder usarla. Podemos crear todas las texturas que necesitemos (y nos permita la memoria), pero sólo podremos usar aquella que esté actualmente vinculada.

Los datos que pasemos a dicha textura (población del algoritmo), se copiarán inicialmente en un buffer de transición, y de ahí se volcarán a la memoria de la tarjeta.

Mediante este buffer podemos comunicarnos con la memoria de la GPU en ambas direcciones, para pasarle datos o para recuperarlos.

4.3.1 Desarrollo de programas Cg

A la hora de escribir el código de los Cgs hay que tener muy presente que los *fragment programs* que se generarán (al compilar los Cgs) se ejecutan de forma paralela. Este concepto, en principio un tanto peregrino para el desarrollador que no ha trabajo nunca con una GPU, se concreta en una serie de puntos a tener en cuenta. En este apartado pretendemos aclarar las implicaciones prácticas que tiene esta característica.

En primer lugar concretaremos el alcance del paralelismo de la ejecución: Los *fragment programs* son programas que se ejecutan sobre ciertos datos de la textura (almacenados en la memoria de la GPU) y, por tanto, todo *fragment program* debe contar con una zona definida donde ejecutarse. La zona de ejecución es lo que se denomina zona de escritura de la textura, y se define mediante comandos OpenGL de especificación de vértices (GlVertex) una vez vinculado y activado el *fragment program* que desea lanzarse (la ejecución comenzará tras cerrarse el bloque de definición de la zona de escritura). El paralelismo de la ejecución se realiza a nivel de píxel, es decir, el código Cg del *fragment program* activado se ejecutará sobre **cada píxel** incluido en la zona de la textura definida como zona de escritura.

Estructura del fichero Cg

Un archivo Cg consiste en un método con cualquier nombre (no debe llamarse necesariamente main, como en C) que, al igual que en C, tiene un tipo de salida y puede recibir parámetros de entrada. Como hemos visto, los procesadores de la GPU funcionan con flujos de datos. El procesador *fragment* opera sobre flujos de píxeles (fragmentos). Las entradas pueden ser de dos tipos (cada uno con su correspondiente sintaxis de declaración que lo identifica):

- Entradas variables: se usan para datos que difieren según el píxel sobre el que se esté ejecutando el *fragment program*. Típicamente se refiere a coordenadas de textura.
- Entradas uniformes: se usan para valores especificados con independencia del flujo principal de datos de entrada, y no cambian con cada elemento del flujo. En nuestro caso, utilizamos este tipo para pasar la textura (memoria de la tarjeta).

Normalmente un Cg necesita leer algún dato para producir el nuevo valor del píxel sobre el que se está ejecutando (en paralelo con el resto de píxel), y para ello se utiliza el comando texRECT, que lee el contenido del píxel de textura correspondiente a las coordenadas dadas como parámetro. Esta instrucción se realiza sobre la textura pasada como entrada uniforme. Las posiciones de los píxeles a leer se introducen como entradas variables.

Los *fragment programs*, además, deben especificar sus salidas como un vector de elementos de un tipo determinado (COLOR). El hardware usa el valor de esta salida

como el color final del fragmento (valor de píxel). Lo habitual es declarar las salidas (normalmente una sola) como parámetros de salida (modificador **out**), y establecer a *void* el tipo de salida del método del programa Cg.

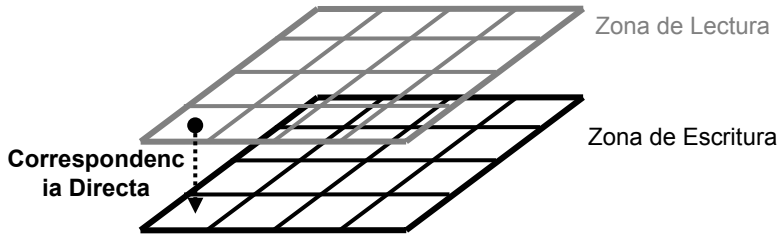
Esencialmente, el cuerpo del fichero consiste en el código Cg necesario para calcular el valor del parámetro de salida (píxel de escritura o de ejecución) deseado.

> ! *El perfil fp30 usado no soporta bucles (ni for ni while) por sí mismo. Sólo pueden usarse si el compilador es capaz de realizar su desenrollado completo (es decir, si el compilador puede determinar el contador de bucle en tiempo de compilación). A fin de lograr la máxima portabilidad, los proyectos desarrollados no incluyen ninguna de estas estructuras [83].*

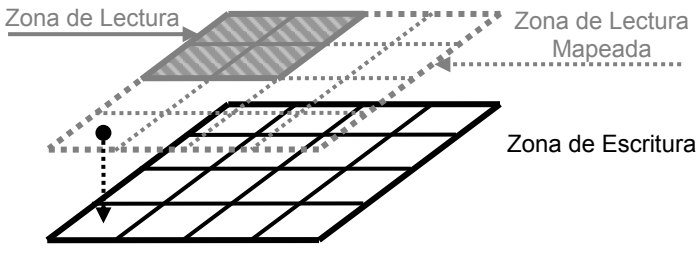
Uso de coordenadas de textura y mecanismo Solapamiento

Si el cálculo del resultado precisa de otros datos almacenados en la textura, han de definirse las zonas de textura donde residen los datos necesarios. A la hora de ejecutarse el *fragment program*, el código puede acceder a los datos de lectura gracias a un mecanismo de solapamiento que realiza la GPU. Para ello se realiza una operación de mapeo de cada una de las zonas de lectura definidas sobre la zona de escritura.

Si ambas zonas tienen las mismas dimensiones, el mapeo es trivial: se solapan ambas zonas haciendo coincidir sus límites y se establece la correspondencia de píxeles en iguales posiciones relativas.



Si las dimensiones o las proporciones de las zonas a solapar difieren, el mapeo se consigue repitiendo u obviando determinados datos según la zona de lectura sea menor o mayor, respectivamente, que la zona de escritura. El modo exacto en que se realiza este ajuste de zonas de textura puede controlarse mediante parámetros en los comandos de definición de texturas.



Coordenadas de textura

La forma de acceder a los flujos de entrada es mediante lecturas de la textura (texRECT) en las posiciones dadas por las coordenadas de textura dadas como variables de entrada. Para cada zona de lectura definida desde OpenGL (en la CPU), se define en el fichero Cg una variable de entrada de algún tipo con significado de coordenada. Cuando se ejecute el *fragment program* sobre cada píxel de escritura, la variable de entrada correspondiente a una determinada zona de lectura contendrá las coordenadas de textura correspondientes al píxel mapeado desde la zona de lectura. Es decir, a cada píxel de escritura le corresponde un píxel de lectura (en cada una de las zonas de lectura) determinado por los mecanismos de solapamiento y mapeo.

Por tanto, para obtener el valor de un píxel del flujo de entrada, se realiza una lectura (texRECT) de la coordenada de textura dada por el valor de la variable de entrada correspondiente a la zona de lectura apropiada.

Es normal que el valor de salida sea función del valor anterior del mismo píxel sobre el que se está ejecutando el *fragment program* (por ejemplo, para el *kernel* de mutación). En estos casos se hace necesario leer el valor antiguo. La forma de hacerlo es simplemente definiendo la zona de escritura también como zona de lectura. Nótese que en este caso las dimensiones siempre coincidirán, y el mapeo será directo.

Definición de las zonas de Lectura

Las zonas de la textura que se utilizarán como flujo de entrada del *fragment program* (zonas de lectura) se definen en el mismo bloque que la zona de escritura. La diferencia entre ambas estriba en el comando OpenGL usado para determinar los vértices que describen cada zona: *glVertex* para la zona de escritura y *glMultiTexCoord* para las diferentes zonas de lectura.

Las zonas de textura son rectangulares, de modo que en su definición se dan cuatro vértices. Para definir el mapeo de las zonas de lectura sobre la zona de escritura se especifican los vértices de las diferentes zonas de lectura que se corresponden con cada uno de los cuatro vértices de escritura. Para poder trabajar simultáneamente con varias zonas de la textura se definen texturas virtuales, lo que llamamos multitexturas, que imitan la existencia de diferentes texturas de las que tomar los datos de entrada. La estructura del bloque de definición de las zonas de textura para tres zonas de lectura sería como sigue:

```
glBegin(GL_QUADS);
    glMultiTexCoord2fvARB( GL_TEXTURE0_ARB, tx0 );
    glMultiTexCoord2fvARB( GL_TEXTURE1_ARB, px0 );
    glMultiTexCoord2fvARB( GL_TEXTURE2_ARB, ax0 );
    glVertex2fv( x0 );
    glMultiTexCoord2fvARB( GL_TEXTURE0_ARB, tx1 );
    glMultiTexCoord2fvARB( GL_TEXTURE1_ARB, px1 );
    glMultiTexCoord2fvARB( GL_TEXTURE2_ARB, ax1 );
    glVertex2fv( x1 );
    glMultiTexCoord2fvARB( GL_TEXTURE0_ARB, tx2 );
    glMultiTexCoord2fvARB( GL_TEXTURE1_ARB, px2 );
    glMultiTexCoord2fvARB( GL_TEXTURE2_ARB, ax2 );
    glVertex2fv( x2 );
    glMultiTexCoord2fvARB( GL_TEXTURE0_ARB, tx3 );
    glMultiTexCoord2fvARB( GL_TEXTURE1_ARB, px3 );
```

```
        glMultiTexCoord2fvARB( GL_TEXTURE2_ARB, ax3 );
        glVertex2fv( x3 );
    glEnd();
```

Definimos una multitextura (con *glMultiTexCoord2fvARB*) con las coordenadas de la zona de lectura (tx0,...,tx3) de modo que para cada vértice de la multitextura le hacemos corresponder el vértice de la zona de escritura (solapamos). A la multitextura se le asigna un nombre de referencia predefinido (de entre los disponibles en OpenGL), en este caso GL_TEXTURE0_ARB. Las instrucciones correspondientes a esta multitextura se han marcado en negrita.

Definimos también una segunda multitextura del mismo modo, esta vez con el nombre GL_TEXTURE1_ARB, e igual para una tercera multitextura (GL_TEXTURE2_ARB).

Cada vez que aparece una instrucción *glVertex*, todas las declaraciones de coordenadas de multitextura precedentes se mapean sobre el vértice de escritura que define.

Referencia en el programa Cg

Las multitexturas se traducen en variables de entrada, coordenadas de textura, en el programa Cg. La forma en que se establece a qué multitextura definida desde OpenGL pertenecen unas determinadas coordenadas, es mediante el tipo con que se declara la variable de entrada: TEXCOORD0 proporcionará el píxel mapeado desde la multitextura definida para GL_TEXTURE0_ARB, y análogamente para el resto de índices.

Uso de parámetros

También es posible declarar parámetros para los programas Cg, y modificar el valor de los mismos.

Desde la CPU se define el parámetro (correspondiente a cierto *fragment program*) asociándole un nombre de referencia, y se establece el valor deseado para el mismo. Este valor se mantendrá constante en tanto no se modifique, y será el mismo para todos los píxeles sobre los que se ejecute el programa.

En el fichero Cg se declara el parámetro con el mismo nombre que se le asoció desde OpenGL, y se usa como cualquier otra variable (uniforme) de entrada.

4.4 Paralelización

Nuestro objetivo es mejorar la eficiencia de la ejecución de un AG sobre nuestro computador, usando para ello la desaprovechada capacidad de cómputo de la tarjeta gráfica del sistema. Por tanto, en nuestro planteamiento la tarjeta gráfica se presenta como un elemento de apoyo al procesador. Esta idea inicial nos llevó a idear una paralelización en la que el proceso principal lanzase un hilo para la GPU, mientras se ejecutaba el algoritmo para CPU.

Sin embargo, la librería OpenGL presenta serias incompatibilidades con las clases de hilos de Windows usadas. Las funciones de las librerías OpenGL sólo pueden ser ejecutadas cuando se ha creado un contexto OpenGL. Este contexto de ejecución es asignado al hilo que lo crea, y como el objeto que implementa el algoritmo genético sobre la tarjeta era creado en el hilo principal de la aplicación, el contexto de OpenGL también se asignaba a dicho hilo. Al crear el hilo secundario, éste no tenía acceso al contexto, por lo que no podía ejecutar las funciones OpenGL. Esto se debe a que Windows parece no permitir que se comparta el contexto entre hilos.

Una solución sería crear un contexto de ejecución OpenGL compartido, de tal forma que cualquier hilo pudiera tener acceso a él. De esta forma podríamos lanzar cada algoritmo genético en sendos hilos secundarios. Sin embargo, descartamos esta posibilidad porque nos parecía superfluo crear dos hilos secundarios, teniendo libre el hilo principal de la aplicación para ejecutar la GPU. Por ello, en la implementación final el proceso principal ejecuta el algoritmo para GPU y el hilo auxiliar ejecuta el algoritmo para CPU.

Como ya mencionamos anteriormente, ambos algoritmos genéticos consumen recursos de CPU. La diferencia entre ellos reside en que el algoritmo genético de CPU usa el procesador para realizar todo el trabajo, mientras que el de la GPU no lo usa en la ejecución de los operadores genéticos, sino solamente para realizar el bucle de generaciones y lanzar la ejecución de los diferentes *fragment programs*.

Aquí es donde reside la paralelización de ambos algoritmos. Mientras el procesador se encarga de ejecutar el algoritmo genético de la CPU, la GPU puede dedicarse a ejecutar un operador genético sobre su población, sin interferir con la CPU.

La aplicación comienza con la creación de los algoritmos genéticos para la CPU y la GPU. Para lograr la paralelización entre ambos creamos un hilo al que asociamos la función de ejecución del algoritmo genético de la CPU. Al lanzar el hilo, el algoritmo genético sobre la CPU comienza su ejecución. En el momento en el que el planificador del procesador lo considere, parará dicho hilo y volverá al hilo principal, que iniciará la ejecución del algoritmo genético sobre la GPU. De este modo, ambos hilos irán turnándose en la posesión del procesador para ejecutar su código.

Como ya comentamos anteriormente, cuando el algoritmo de GPU esté ejecutando un operador genético, el procesador queda desocupado por dicho hilo, y puede pasar a ser usado por el hilo del algoritmo de CPU, haciendo paralelos ambos algoritmos. Además, mientras la GPU está ejecutando un operador genético, el procesador puede adelantar trabajo ejecutando instrucciones del algoritmo genético de la GPU, como la definición de flujos para el siguiente operador.

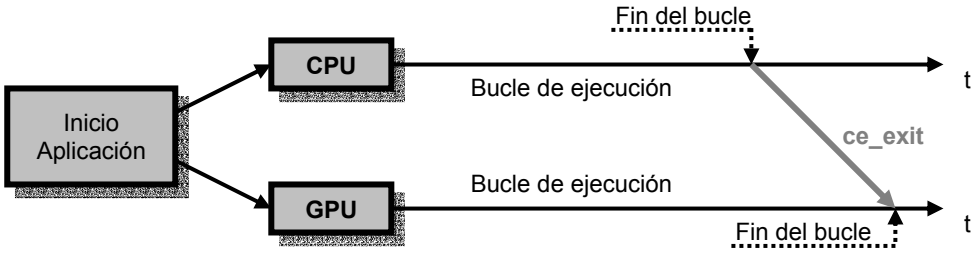
Debido a que consideramos a la GPU una ayuda al algoritmo de CPU, es este último el que determina la conclusión de la aplicación, cuando termine de ejecutar el número de generaciones asignado por el usuario. En dicho momento, lanzará un evento de fin de trabajo, que es chequeado en cada generación por el algoritmo de GPU. Cuando dicho evento sea señalado, el algoritmo de GPU deja de realizar su bucle principal, terminando así la ejecución del mismo:

CPU

```
while (not condicion_parada)
   bucle_ejecución;
::SetEvent(ce_exit);
```

GPU

```
while (not SetEvent(cpu->ce_exit)
   bucle_ejecución;
```



Esquema de parada del AGP

4.4.1 Migración

La migración supone una comunicación entre algoritmos genéticos, que consiste en el paso de individuos entre las poblaciones de ambos. Para ello se implementa una jerarquía de clases en la que se define una clase padre, con las características propias de la migración, de la que derivan dos clases de algoritmos genéticos, cada una implementada según el procesador en el que serán ejecutados (CPU o GPU).

El paso de esta información entre algoritmos supone un coste de comunicación adicional al proceso global, puesto que hay que convertir los datos a las estructuras específicas de cada algoritmo, así como tomar los datos de la población de la memoria de cada uno de ellos.

Por esto, se optó por una estructura de datos simple para este intercambio. Esta estructura consiste en un array de floats, con los datos de los individuos de la población de intercambio. Esto nos permite una fácil adaptación a cada una de las estructuras de los algoritmos: el algoritmo de CPU sólo tiene que recorrer el array tomado y copiar esta información en los genes de sus individuos descartados, mientras que el algoritmo de GPU sólo tiene que transformar los datos a un array de píxeles y copiarlos en la textura de la tarjeta.

Esto nos permite intercambiar los individuos entre los algoritmos sin tener que crear nuevos objetos dinámicos, lo que nos supondría una pérdida de tiempo mayor.

El algoritmo genético transforma la población de intercambio a la estructura de datos simple antes comentada, que será leída por el algoritmo genético al que migrarán los individuos, transformándola a sus estructuras de datos propias. Para poder leer la población intermedia de un algoritmo, el otro dispone de la dirección de memoria del primero (puntero).

A la hora de intercambiar un individuo, nos basta con pasar la información correspondiente a su genotipo, ya que a partir de éste se puede calcular su fitness. Sin embargo, decidimos pasar también el fitness de cada individuo junto con su genotipo,

ya que nos suponía un gasto ínfimo comparado con el coste de volver a evaluar los individuos una vez migrados en la nueva.

Cada algoritmo genético tiene acceso a su propia memoria, de donde tomará los individuos a migrar. El número de migrados dependerá del tamaño de la población, siendo aproximadamente el 4%:

$$Intercambio = \frac{tam\_pob}{25} + 1$$

Este porcentaje de individuos migrados se toma de las primeras posiciones de la población. Esto se debe a la particularidad de la élite.

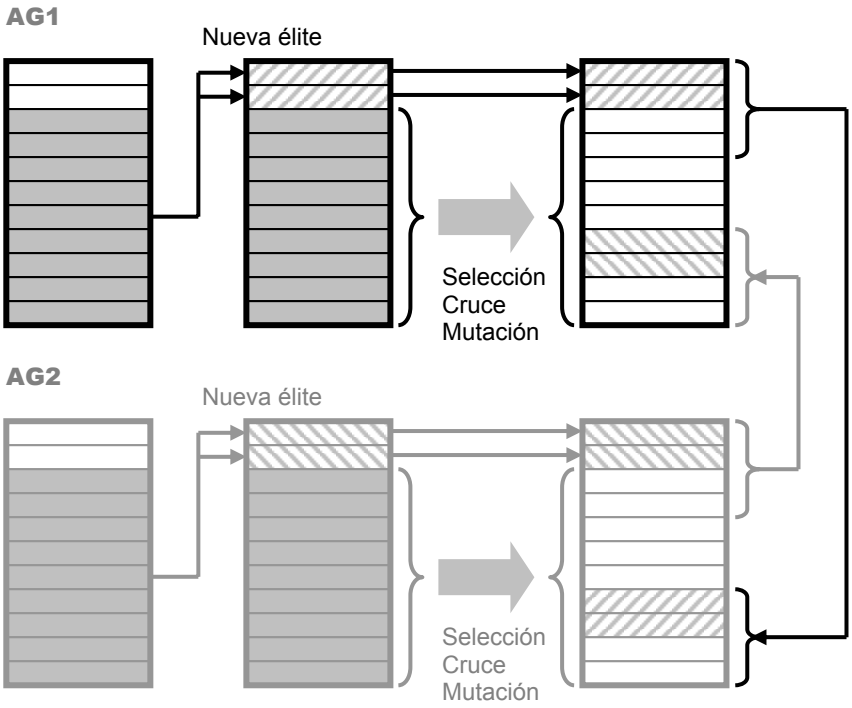
El mejor individuo de la población se busca en cada generación, al comienzo del proceso de selección. Este individuo, como ya comentamos, se copia en las dos primeras posiciones de la población, quedando invariante hasta el final del bucle de ejecución (no muta).

La consecuencia de esto es que, al comienzo de la siguiente generación, los dos primeros individuos de la población corresponden a la élite de la generación anterior. Esto no significa que sean los mejores en ese instante, ya que pueden haber aparecido mejores individuos resultantes del cruce y mutación del resto de la población en la generación pasada. Sin embargo, por lo menos sabemos seguro que están dentro de los mejores.

De esta manera nos aseguramos de que, a la hora de migrar, migramos al menos uno de los mejores individuos de la población, sino el mejor.

La información de los individuos migrados se pasa a la población del algoritmo tomador. Como ya comentamos, usamos esta información para modificar los individuos a desaparecer de la población. Estos son los últimos de la población.

La decisión de desechar estos individuos no es justa, pero nos asegura una pérdida de tiempo mínima en la transacción de la información. Además, como en cada ejecución del bucle la élite de la generación anterior se sitúa en las primeras posiciones de la población, desechar las últimas nos asegura no perder algunos de los mejores individuos de la población.



Esquema de transferencia de datos en la migración

El número de migraciones entre los algoritmos es suministrado por el usuario, y la aplicación se encarga de calcular las generaciones en las que deben realizarse.

Conviene destacar que las migraciones deben estar balanceadas dentro de la aplicación, es decir, que se realicen en intervalos iguales y que la primera y última no estén muy próximas al inicio y fin de la aplicación. En el primer caso, generación próxima al inicio, los individuos intercambiados no proporcionarían diversidad suficiente, puesto que no habría dado tiempo a que convergieran a ninguna solución parcial. En el segundo, la migración sería inútil, próxima al fin, puesto que no daría tiempo a que la diversidad introducida calase en la nueva población.

Para evitar estos dos casos, utilizamos la fórmula de cálculo:

$$Frecuencia = \frac{generaciones\_CPU}{migraciones + 1} + 1$$

Como ya comentamos en la introducción de este capítulo, se ha optado por una migración asíncrona frente a la migración síncrona. Esto deriva de la motivación general de este trabajo.

Queremos usar la capacidad de cómputo de la tarjeta para adelantar trabajo del algoritmo. Si consideramos la aproximación síncrona, ambos algoritmos migrarían en la misma generación. En el momento en que uno de los dos tomara ventaja sobre el anterior, tendría que esperar a que aquél llegase a la generación de intercambio. De esta manera estaríamos desaprovechando el rendimiento del mejor algoritmo, al tenerlo parado.

En el caso asíncrono, uno de los dos algoritmos marca la pauta de migración, mientras que el otro espera a que le indiquen el momento de migrar. Como consideramos a la GPU una ayuda, es el algoritmo CPU quien controla la migración e indica el momento de realizarla. De esta manera, el algoritmo de GPU podrá realizar las generaciones que le den tiempo antes de que el algoritmo de CPU pida la migración. Considerando más rápido al algoritmo de GPU, dadas las características paralelas de la tarjeta gráfica, la conclusión lógica es que éste haga evolucionar su población más generaciones de lo que lo haría la CPU, lo que supone una posible mejora en la solución parcial.

Para realizar la migración asíncrona, los AG's disponen de eventos que lanzan en el momento oportuno para comunicarse entre ellos. Estos eventos indican el inicio y fin de migración (sólo usados por el algoritmo de CPU) y de listo para el intercambio (usados por ambos algoritmos).

Cabe destacar la necesidad de estos dos últimos eventos (listo y fin). Cuando se entra en la fase de migración, cada algoritmo debe definir su población de intercambio. Cada uno de ellos sólo podrá acceder a la población de intercambio generada por el otro cuando aquél la haya definido, si no podríamos tomar datos incorrectos. Para evitar esto se usan los eventos anteriores, que indican al algoritmo receptor que la población de intercambio a tomar ya está disponible (evento *listo*), y al algoritmo donador que puede salir de la fase de migración (evento *fin* de migración) y no hay peligro de corromper los datos de intercambio, puesto que ya han sido copiados por el tomador y no se necesitan.

El funcionamiento de la aplicación con algoritmos que migran es básicamente el mismo que sin ella. Sin embargo, cuando el algoritmo de CPU llegue a una generación de intercambio, lanzará el evento de inicio de la fase de migración. Este evento es chequeado por el algoritmo de GPU en cada paso del bucle de ejecución. Cuando el evento sea señalado, la GPU entrará en fase de migración.

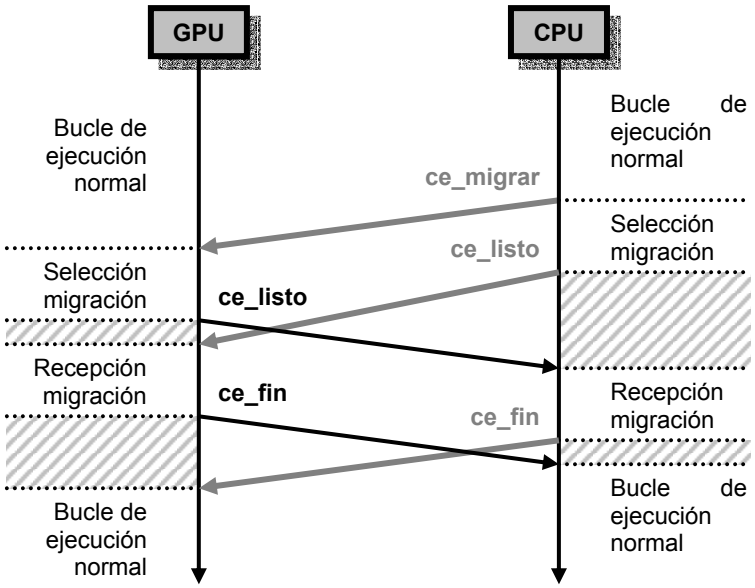
Cuando ambos algoritmos entran en dicha fase, definen sus poblaciones de intercambio. Cuando un algoritmo termina dicha definición lanza el evento de listo para el intercambio, y espera a que el otro le lance el mismo. Una vez que éste es recibido, pasa a copiar la información (población intermedia vecina) en su propia población, descartando el mismo número de individuos tomados. Después lanza el evento de fin de migración y espera a recibir el mismo evento, del otro algoritmo, que le indica que el intercambio ha concluido y que puede seguir su ejecución normal. Este comportamiento se repetirá para cada generación de intercambio establecida por la formula anteriormente descrita:

CPU

```
while (not condicion_parada){
  if (toca_migrar(generacion)){
    ::SetEvent(ce_migrar);
    selección_migracion();
    ::SetEvent(ce_listo);
    ::WaitForEvent(gpu->ce_listo);
    recibir_migracion();
    ::SetEvent(ce_fin);
    ::WaitForEvent(gpu->ce_fin);
  }
  bucle_normal_ejecución();
}
```

GPU

```
while (not SetEvent(cpu->ce_exit)){
  if (SetEvent(cpu->ce_migrar)){
    selección_migracion();
    ::SetEvent(ce_listo);
    ::WaitForEvent(cpu->ce_listo);
    recibir_migracion();
    ::SetEvent(ce_fin);
    ::WaitForEvent(cpu->ce_fin);
  }
  bucle_normal_ejecución();
}
```



Esquema de ejecución paralela de la migración

4.5 Solución al problema defectivo

Como parte del proyecto, se ha implementado la solución al problema defectivo comentada en el capítulo anterior: el mapeo evolutivo.

Recordamos que, en la solución propuesta, cada individuo está formado por dos cromosomas: uno de datos (binario) y otro de mapeo del genotipo en el fenotipo (números enteros). Además, se introducen algunos cambios en los operadores genéticos.

El bucle de evolución es igual al de todos los AGs implementados:

```
generación de la población inicial
evaluación de la población inicial
while no se cumpla la condición de parada do
   selección
   cruce
   mutación
   evaluación de los individuos
end while
```

A continuación describimos los cambios introducidos en las estructuras de datos usadas, así como en la implementación de los diferentes pasos del proceso de evolución.

4.5.1 Estructuras de datos

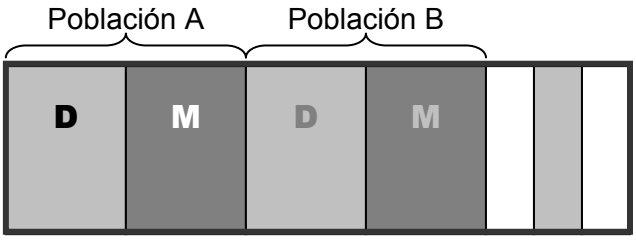
Los individuos consisten ahora en dos vectores de genes, en lugar de en uno solo, ambos de longitud igual al número de genes del individuo: uno de ellos almacena los genes del individuo (vector binario) y el otro almacena las posiciones de mapeo (vector de enteros en [0, número_genes-1]).

Implementación en la CPU

La implementación es inmediata: la clase Individuo añade un nuevo elemento del mismo tipo que el cromosoma de datos para almacenar los valores de mapeo.

Implementación en la GPU

Los valores de mapeo se almacenan a continuación de los genes de cada individuo, de modo que la zona de genes ocupa ahora el doble que en las implementaciones anteriores. Si llamamos D a la zona de datos, y M a la zona de mapeo, el esquema de zonas de la textura queda ahora:



4.5.2 Evaluación de los individuos

Esta aproximación al problema de las funciones defectivas utiliza un mapeo del genotipo en el fenotipo a evaluar. Por tanto, en vez de aplicar la función trampa directamente sobre el cromosoma de datos (genes) del individuo para obtener su fitness, debemos realizar previamente una operación de muestreo para construir el fenotipo, y evaluar entonces la cadena fenotípica formada con la ftrap.

Ejemplo

Para un individuo con los siguientes cromosomas:

Cromosoma de datos	1	1	0	1	1	1	0	0
Cromosoma de mapeo	7	6	0	1	2	3	7	7

Sea la función ftrap con K=4 que asigna 3 a las cadenas sin 1s, 2 a las cadenas con un 1, 1 a las cadenas con dos 1s, 0 a las cadenas con tres 1s, y 4 a las cadenas con cuatro 1s.

Para el individuo dado no se aplica sobre la cadena 11011100 (que resultaría en un fitness de 0 + 1 = 1), sino que primero se mapea el cromosoma de datos al fenotipo 00110100 y luego se aplica la ftrap para obtener el fitness: 1 + 2 = 3.

■

Para construir el fenotipo se recorre secuencialmente el cromosoma de mapeo, leyendo las posiciones del cromosoma de datos indicadas. Una vez realizado este paso previo, se realiza la evaluación habitual definida por la función:

$$ftrap(unos) = \begin{cases} K - unos - 1, & si\ unos < K \\ K, & si\ unos == K \end{cases}$$

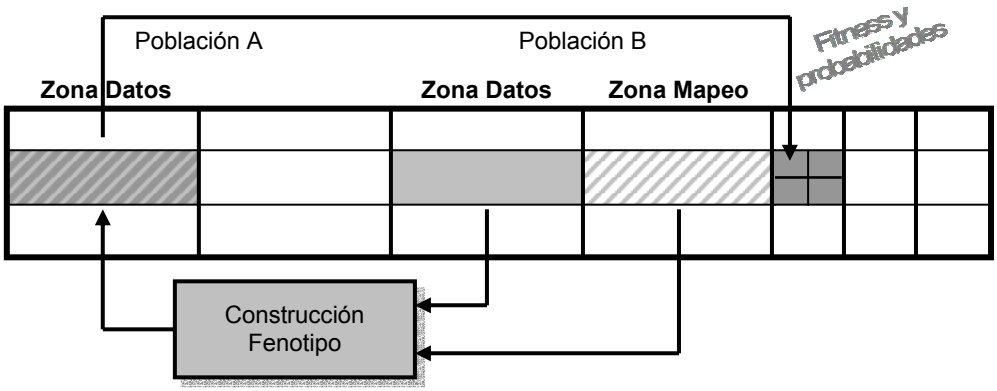
Implementación en la CPU

Utilizamos un vector auxiliar de enteros para construir el fenotipo. Recorremos para ello el cromosoma de mapeo secuencialmente, insertando en el vector auxiliar el valor almacenado en la posición indicada, por el gen de mapeo, del cromosoma de datos. Después calculamos el fitness del vector auxiliar (fenotipo) dado por la función defectiva, en el modo habitual.

Implementación en la GPU

Para realizar el paso previo de la construcción del fenotipo debemos mapear adecuadamente el genotipo de cada individuo. Como hemos visto, esta operación consiste básicamente en recorrer secuencialmente el cromosoma de mapeo, leyendo los correspondientes genes y volcándolos en el fenotipo final. Por tanto, la construcción del fenotipo se realiza mediante un *fragment program* auxiliar que utiliza dos zonas de lectura: la correspondiente a los cromosomas de datos y la correspondiente a los cromosomas de mapeo, ambas en la Población B, y como zona de escritura define la zona de cromosomas de datos de la Población A. El volcado de fenotipos se realiza sobre la zona de datos del área de escritura de la textura, para no perder el genotipo (que permanecerá intacto en el área de lectura). Una vez actualizada la textura, se ejecuta el *fragment program* de evaluación correspondiente a la ftrap (el mismo usado en la implementación sin mejora) estableciendo ahora la zona de lectura de genes a la correspondiente al fenotipo (buffer de escritura), en lugar de establecerla a la del genotipo, como hacíamos antes.

Nótese que, aunque se ha introducido un *fragment program* auxiliar, los flujos de entrada y de salida continúan siendo los habituales para el resto de *fragments programs* (excepto para la zona de de lectura correspondiente a los cromosomas de datos para la evaluación). De este modo, los fenotipos se perderán al ser sobrescritos por el siguiente *kernel*, pero no importa puesto que el valor de fitness ya está calculado correctamente (sobre el fenotipo) y el fenotipo no vuelve a usarse hasta la siguiente evaluación, en la que habrá que construirlo.



4.5.3 Cruce de los individuos

El operador de cruce no sufre ningún cambio en sí mismo, simplemente se aplica dos veces: una vez sobre el cromosoma de datos y otra sobre el cromosoma de mapeo, de forma habitual. Así logramos que ambas operaciones sean independientes.

4.5.4 Mutación de los individuos

El método de mapeo evolutivo introduce en este paso un nuevo operador: reemplazamiento de genes, mediante el cual un gen del cromosoma de mapeo puede ser copiado a otro gen.

La mutación se implementa del modo habitual para el cromosoma de datos, y para el cromosoma de mapeo hemos unido los efectos del operador tradicional de mutación y del nuevo operador de reemplazamiento simplemente generando un entero en [0, número_genes – 1] cuando hay que mutar un gen de mapeo. Para ello se mapea el *fragment program* habitual sobre la zona de datos, y un nuevo *fragment program* (que trabaja con enteros en vez de con binarios) sobre la zona de mapeo. De este modo, al adoptar los genes de mapeo cualquier valor en el intervalo, es posible introducir repeticiones en el cromosoma de mapeo (copias de genes de mapeo).

5 Resultados experimentales

Dado que el objetivo del presente trabajo es mejorar el rendimiento de AGs ejecutados sobre sistemas monoprocesadores, los experimentos realizados se han centrado en los tiempos de ejecución logrados y en la comparación de la ejecución de los AGs sobre la CPU en el modo tradicional frente a la ejecución en paralelo también sobre la GPU.

Los AGs implementados resuelven tres problemas de prueba ampliamente tratados en la literatura. No es nuestro objetivo mejorar la calidad de las soluciones, sino solamente utilizarlos como un parámetro contrastado en las medidas de speed-up y eficiencia. Por el mismo motivo tampoco hemos estudiado la bondad de la solución implementada para mejorar los resultados con funciones de aptitud defectivas (algoritmo de mapeo evolutivo) [19]. Sin embargo se han realizado pruebas de medición sobre los valores de fitness obtenidos, a fin de comprobar el correcto funcionamiento de nuestras implementaciones.

Las mediciones que realmente nos interesa para comprobar la consecución de los objetivos fijados se refieren a la relación de tiempos de ejecución (*speed-up*) obtenidos para una implementación tradicional frente a la implementación paralela propuesta en el presente trabajo, y cómo depende esta ganancia de los diferentes parámetros involucrados en el AG. Los parámetros con los que hemos jugado, los que pueden repercutir en el tiempo de ejecución del algoritmo, son:

- Tamaño de población: determina el número de individuos de cada población, es decir, la altura de la textura empleada en la GPU. Puesto que el paralelismo en la evaluación de individuos se realiza por columnas, es de esperar que cuanto mayor sea el tamaño de la población, mayor rendimiento obtendremos en la GPU.
- Número de generaciones: determina el número de generaciones a efectuarse en la CPU (la GPU realizará las que le dé tiempo).
- Longitud del cromosoma: determina el número de genes de cada individuo, es decir, la anchura de la textura empleada en la GPU. Puesto que el paralelismo en la evaluación de individuos se realiza por columnas es de esperar que, cuanto mayor sea la longitud del cromosoma, menor rendimiento obtendremos en la GPU.

Así, fijando el resto de parámetros a valores por defecto, se han tomado medidas de tiempos de ejecución para diferentes valores de cada uno de los parámetros, dejando patente la influencia de cada uno de los factores en el rendimiento.

➔ *A fin de automatizar la ejecución de grandes baterías de pruebas, se ha desarrollado un sistema de especificación y recogida de mediciones mediante ficheros XML. La especificación de éstos, así como el modo de usarlos, se incluye en el Apéndice I.*

En el siguiente apartado se incluyen los resultados medios obtenidos. Si el lector desea ver los datos concretos de las diferentes baterías de pruebas realizadas, le remitimos al los ficheros .xml de baterías de pruebas incluidos en el CD-ROM adjunto.

5.1 Rendimientos

El speed-up se define como la mejora de velocidad obtenida al aplicar una cierta mejora M que permite ejecutar una parte del código x veces más rápido. Puede calcularse fácilmente como el cociente del tiempo de ejecución sin mejora y el tiempo de ejecución con mejora:

$$Speed-up(E) = \frac{T_{\text{ejec sin M}}}{T_{\text{ejec con M}}}$$

A la hora de definir el rendimiento es importante asegurar la igualdad de condiciones entre los dos modos de ejecución a comparar. Esto nos llevó a plantear inicialmente la paralelización del AG de un modo poco eficiente: tanto la GPU como la CPU realizaban el mismo número de generaciones (las especificadas por el usuario) y, una vez alcanzadas éstas en cualquiera de los hilos, se esperaba a que el otro proceso acabase. Sin embargo este planteamiento, aunque equitativo, no es el más indicado para nuestros propósitos. Dado que la GPU actúa como elemento auxiliar de la CPU, el imponer que la primera alcance un límite fijo malogra el aumento del rendimiento potencial. Tanto si la GPU resulta más lenta que la CPU como si resulta más rápida (opción esperada), no estamos obteniendo el resultado óptimo: si el tiempo de GPU es el mayor, la CPU terminará y tendrá que esperar a su *ayudante* sin hacer nada; si el tiempo de GPU es menor, una vez terminado el trabajo asignado, caeremos de nuevo en la situación que nos planteamos subsanar (inutilización de la GPU).

A fin de lograr el mejor speed-up posible, las implementaciones finales llevan a cabo una migración asíncrona en la que la CPU planifica los eventos de migración y la terminación del AG. Así, mientras la CPU no alcance el número de generaciones prefijadas, la GPU estará utilizando su capacidad de cómputo, y cuando la CPU termine se finalizará el algoritmo global.

El aprovechamiento de la GPU es máximo, pero nos queda por resolver el problema del cálculo justo del aumento de rendimiento obtenido. Para una batería de pruebas correspondiente a los parámetros del AG global *tam_pob* (tamaño de población), *num_geners* (número de generaciones) y *lcrom* (longitud de cromosomas), definimos a continuación a qué nos referimos con *tiempo de ejecución sin mejora* y *tiempo de ejecución con mejora:*

- **Tiempo de ejecución con mejora**: es el tiempo que el AG paralelo tarda en ejecutar sobre la CPU *num_geners* generaciones sobre una población de *tam_pob*/2 individuos con *lcrom* genes cada uno; y sobre la GPU las generaciones que dé tiempo a hacer (en general diferente de *num_geners*) sobre una población de *tam_pob*/2 individuos con *lcrom* genes cada uno.

- **Tiempo de ejecución sin mejora**: es el tiempo que el AG secuencial tarda en ejecutar únicamente sobre la CPU *num_geners*+ ***num_geners_GPU*** generaciones sobre una población de *tam_pob* individuos con *lcrom* genes cada uno.

A la vista de las definiciones anteriores resulta evidente que las ejecuciones sin mejora deben realizarse después de las del algoritmo mejorado, puesto que la salida ***num_geners_GPU*** se utiliza como parámetro de entrada del AG ejecutado sin mejora.

Resultados obtenidos

Para los experimentos realizados, se han fijado las probabilidades de aplicación de los operadores genéticos a los valores habituales, y el número de migraciones realizadas a lo largo de todo el proceso a 5.

Los parámetros de los AGs se varían individualmente para constatar el efecto de cada uno sobre el rendimiento obtenido, fijando el resto a los siguientes valores por defecto:

- Probabilidad de mutación: 0.02
- Probabilidad de cruce: 0.80
- Nº de migraciones total: 5
- Nº de individuos migrados: (tam_pob/25) + 1

Valores por defecto

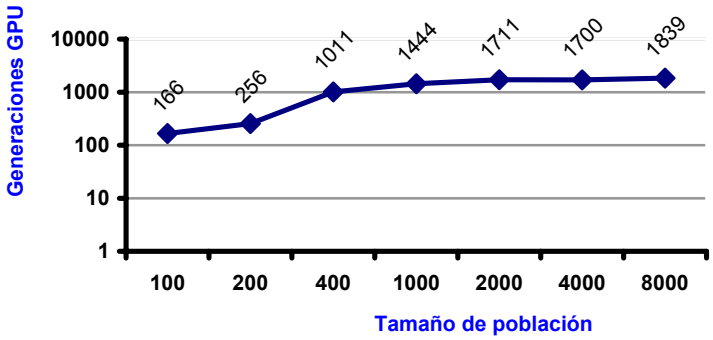
- Tamaño de población: 1000
- Nº de generaciones: 500

A continuación se incluyen los resultados **medios** (para bloques de 10 pruebas) obtenidos para los diferentes problemas planteados, así como sus gráficas correspondientes. Las medidas de tiempos están expresadas en **microsegundos**.

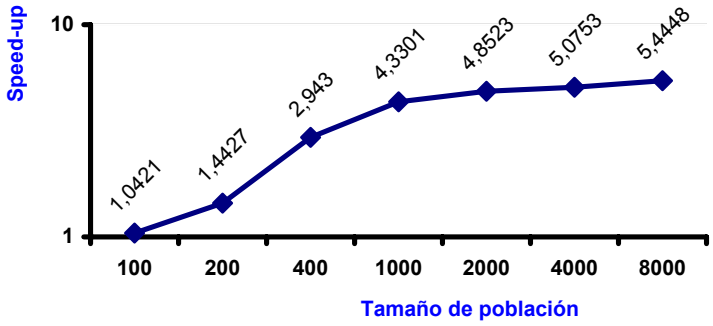
5.1.1 Onemax

Tamaño de población

Tamaño población	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
100	166	1255111,86	1204390,64	**1,0421136**
200	256	2894940,40	2006560,44	**1,4427377**
400	1011	11749873,01	3992510,60	**2,9429785**
1000	1444	40782692,73	9418364,98	**4,3301245**
2000	1711	94853827,49	19548132,05	**4,85232181**
4000	1700	194991220,23	38419634,77	**5,07530125**
8000	1839	444290695,03	81599207,40	**5,4447918**



Puesto que la CPU trabaja secuencialmente, el aumento del tamaño de la población incrementará proporcionalmente el tiempo de ejecución del AG correspondiente. Cuanto más tarde la CPU en completar el número de generaciones programadas, de más tiempo dispondrá la GPU para trabajar. Como el tamaño de población incrementa el tiempo de ejecución de la GPU mucho más despacio (y sólo porque arrastra cálculos secuenciales sobre la CPU), el número de generaciones que logra realizar la tarjeta crece rápidamente con el número de individuos.



Los resultados obtenidos en tiempos de ejecución confirman nuestras expectativas iniciales de una mejora de speed-up en función del tamaño de la población. El rendimiento asciende casi exponencialmente hasta alcanzar un umbral (cercano a los 500 individuos de población total) a partir del cual el crecimiento se hace lineal.

La clara influencia del número de individuos en la mejora del speed-up se debe al carácter paralelo de la GPU. Como vimos en el capítulo de implementación y desarrollo, la evaluación de los individuos se realiza sobre la GPU por columnas (sólo se produce paralelismo en esta dimensión), y el número de elementos que forman una columna depende directamente del tamaño de la población (recordamos que hay una fila por individuo); en consecuencia, el aumento de la población produce un aumento del paralelismo de la evaluación y la subsiguiente mejora del rendimiento.

Cabría esperar que los aumentos de ambos factores (generaciones GPU y speed-up) con el tamaño de la población tuvieran forma lineal. Sin embargo esta tasa de crecimiento no se alcanza hasta un tamaño en torno a los 500 individuos, apareciendo un ligero valle para valores menores a este umbral. Creemos que este fenómeno puede deberse a la planificación de hilos realizada por el sistema operativo, y a la tecnología Hiper-threading con la que cuenta el procesador del sistema usado. El efecto de ambos

fenómenos hace que se dé prioridad al hilo con mayor carga de trabajo (CPU) y que, a medida que éste necesita hacer más accesos a memoria, se vayan creando más huecos en los que el otro proceso puede tomar el control para proseguir su ejecución. Hasta que los recursos de memoria se saturan, el proceso CPU puede ejecutarse fluidamente y, puesto que goza de la prioridad del planificador, no produce oportunidades de ejecución para el proceso GPU. Sin embargo, cuando se alcanza cierto umbral, los fallos de memoria generan huecos en la ejecución de este proceso, brindando al proceso GPU la posibilidad de ejecutarse.

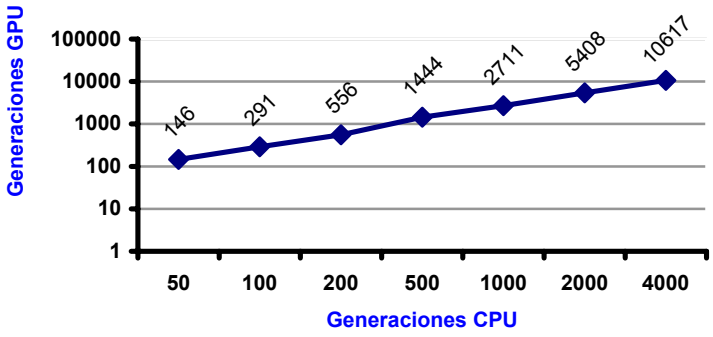
*La tecnología **Hyper-Threading** es un novedoso diseño de Intel que permite que las aplicaciones de software multi-threaded (multi proceso) ejecuten varios subprocesos en paralelo dentro de cada procesador, dando como resultado una mayor utilización de los recursos del procesador. Resumiendo, consiste en colocar dos procesadores lógicos en una única oblea de CPU. Como resultado, una mejora media de un 10-30% en la utilización de recursos de la CPU (según la documentación de Intel), origina una capacidad de procesado mayor.*

El rendimiento obtenido varía dentro de los valores indicados dependiendo de la aplicación y la situación. Supongamos que se está ejecutando un hilo en el primer procesador lógico. Este hilo estará consumiendo recursos (unidades de ejecución, cache, y posiblemente acceso a memoria a través del bus) tan eficientemente como sea posible ejecutar este hilo particular. Al mismo tiempo, se planifica un segundo hilo al segundo procesador lógico, que también necesitará acceder a las unidades de ejecución, a la cache e incluso al bus. Surge entonces un conflicto, pudiendo pasar dos cosas [88]: el hilo inicial que se ejecuta en el primer procesador lógico puede reducir su velocidad, o bien el hilo que se ejecuta en el segundo procesador lógico puede ver limitado su rendimiento.

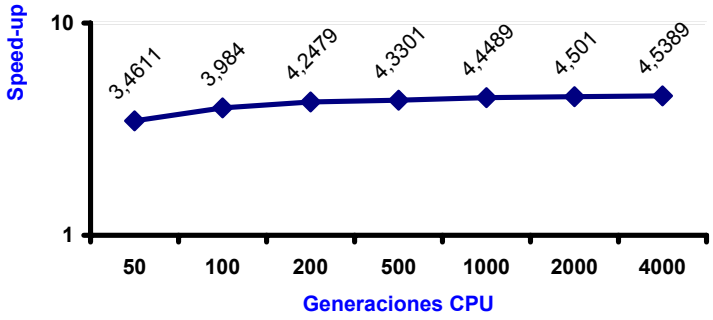
Así, parece que hasta que las cargas de trabajo para ambos hilos no alcanzan un volumen crítico, la CPU goza de mayor prioridad para realizar sus operaciones frente a la GPU.

Número de generaciones

Generaciones CPU	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
50	146	4464162,82	1289823,58	**3,4610647**
100	291	8614729,95	2162349,78	**3,9839669**
200	556	16232332,62	3821218,38	**4,2479469**
500	1444	40782692,73	9418364,98	**4,3301245**
1000	2711	76193514,88	17126494,51	**4,44886809**
2000	5408	151691612,33	33701549,17	**4,50102788**
4000	10617	301349750,98	66392999,23	**4,53887841**



El número de generaciones GPU que logra el algoritmo mejorado crece de modo lineal cuando aumenta el número de generaciones fijadas para la CPU, como era de esperar. Debido a que el número de generaciones no influye en la carga de trabajo de una iteración del bucle evolutivo, el número de generaciones que la GPU consigue efectuar es siempre proporcional al número de generaciones CPU.

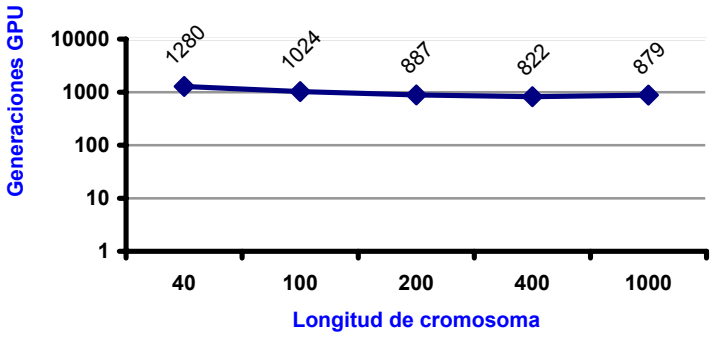


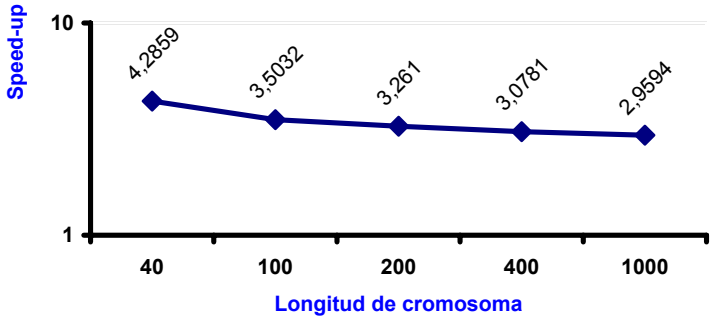
El speed-up obtenido no varía apenas conforme se aumenta el número de generaciones a realizar por el algoritmo CPU. Este fenómeno responde a que el bucle de generaciones se realiza sobre la CPU, y por tanto no se ve favorecido por el carácter paralelo de la GPU.

Longitud de cromosoma

Puesto que el paralelismo de la GPU se explota sólo en la dimensión de las columnas, para la evaluación de los individuos, es lógico que descienda la mejora obtenida según aumenta la anchura de la textura manejada (número de genes de cada individuo). Por tanto es conveniente emplear cromosomas de poca longitud.

Nº de genes	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
40	1280	26332348,02	6143916,98	**4,28592185**
100	1024	33660429,74	9608499,18	**3,50319327**
200	887	48139473,36	14762175,42	**3,26100130**
400	822	25689213,76	79074652,18	**3,07812660**
1000	879	202850227,52	68544162,87	**2,95940922**

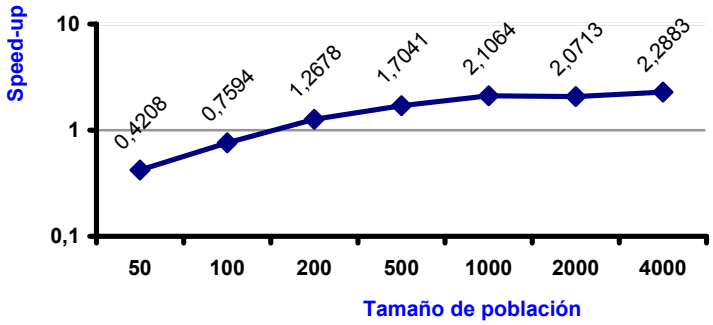




CPU vs GPU

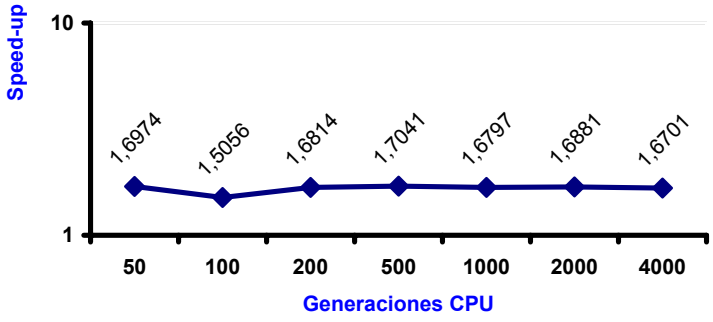
Creemos interesante incluir también la comparativa entre ejecuciones sobre un único hilo, demostrando así que efectivamente la GPU trabaja más eficientemente que la CPU en la ejecución de AGs. Téngase en cuenta que sólo se ejecuta una parte del AG sobre la GPU, y el resto sobre la CPU. Si bien la evaluación representa la mayor parte del tiempo de cómputo, recordamos que ésta se ejecuta paralelamente sólo en cada columna (se envía una columna a la GPU cada vez). En este caso definimos el speed-up como T_{CPU}/T_{GPU}.

Tamaño población	T CPU	T GPU	Speed-up
50	474188,00	1126988,85	**0,4207566**
100	957633,81	1260979,44	**0,7594365**
200	1939996,19	1530216,15	**1,2677923**
500	5131182,27	3011132,52	**1,7040706**
1000	11380053,18	5402490,09	**2,10644591**
2000	23100355,36	11152743,79	**2,07127105**
4000	49873269,97	21794686,58	**2,28832242**



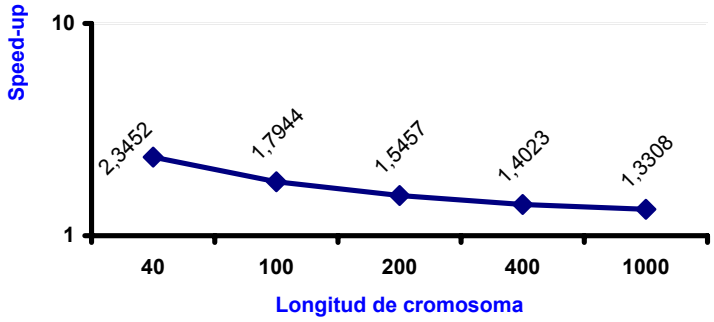
El speed-up aumenta de forma aproximadamente lineal puesto que, efectivamente, el cociente entre los tiempos de ejecución del algoritmo sin mejora y el mejorado debe crecer linealmente con el tamaño de la población. Puesto que el algoritmo puramente CPU es secuencial, el tiempo de ejecución crecerá proporcionalmente con el volumen de datos a tratar. Por el contrario, considerando que el tiempo de ejecución se debe en su mayor parte al tiempo de evaluación, y teniendo en cuenta que para el algoritmo mejorado la evaluación se realiza en paralelo (cuesta siempre lo mismo independientemente del número de individuos de la población), es de esperar que el tiempo de ejecución del algoritmo sobre GPU tenga forma aproximadamente constante en el tamaño de población. Se observa un estancamiento a partir de poblaciones cone 1000 individuos, debido seguramente a que a partir de dicho umbral cobran mayor peso las partes del algoritmo que se ejecutan sin mejora (sobre la CPU).

Generaciones	T CPU	T GPU	Speed-up
50	567390,47	334274,02	**1,6973813**
100	1058432,26	703012,71	**1,5055663**
200	2064895,83	1228112,51	**1,6813572**
500	5131182,27	3011132,52	**1,7040706**
1000	9980601,96	5942069,76	**1,67965075**
2000	19857009,58	11763257,27	**1,68805367**
4000	39390961,16	23586644,50	**1,67005363**



Puesto que el aumento de generaciones a realizar no se ve beneficiado por el uso de la GPU, se obtiene una proporción de tiempos de ejecución prácticamente constante, resultando que el algoritmo mejorado es casi 1,7 veces superior al algoritmo sin mejora.

Nº genes	T CPU	T GPU	Speed-up
40	3322524,37	1416755,83	**2,3451637**
100	5403627,83	3011329,44	**1,7944326**
200	8306997,76	5374356,60	**1,545673**
400	14599739,86	10411233,22	**1,4023065**
1000	37877210,69	28462647,67	**1,33076905**



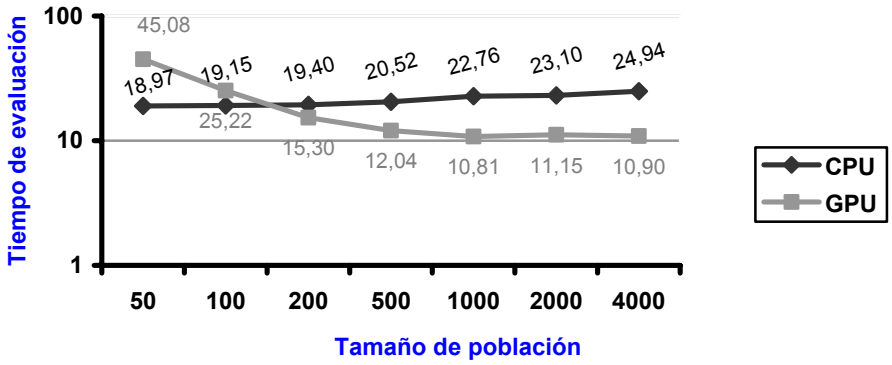
La mejora GPU se hace más patente cuando mayor sea el paralelismo logrado. Por tanto, dado que un aumento en la anchura de la textura produce un aumento del cálculo necesario, pero no un aumento de paralelismo, es claro que es preferible resolver problemas que usen longitudes de cromosoma pequeñas.

Esfuerzo computacional

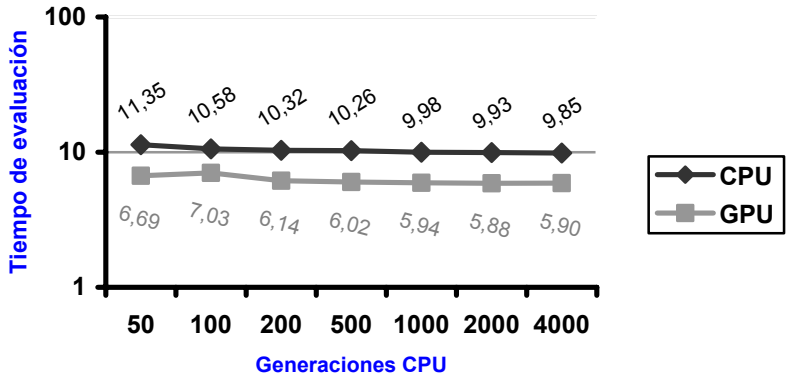
Consideramos la medida del esfuerzo computacional como el tiempo aproximado que se dedica a cada individuo (en el proceso evolutivo completo) y que se corresponderá aproximadamente con el tiempo dedicado a cada evaluación:

$$T_{evaluación} = \frac{T_{ejecución}}{Tam\_pob \cdot Geners_{total}}$$

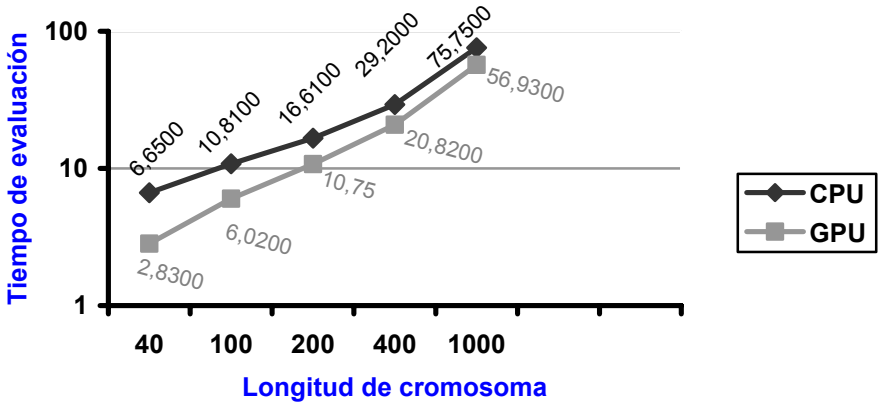
A continuación mostramos las gráficas comparativas de esta magnitud para los algoritmos con mejora y sin mejora, variando cada uno de los parámetros de los AGs.



Con esta gráfica queda patente el motivo por el que se obtienen tan buenos resultados: mientras que el esfuerzo de cómputo viene a ser constante para la CPU, con un ligero aumento a medida que crece el número de individuos, en el algoritmo mejorado disminuye casi exponencialmente con el tamaño de la población. Parece que el esfuerzo computacional se estabiliza cerca de 11 microsegundos para el algoritmo que usa la GPU.



Puesto que un aumento en el número de generaciones CPU no se ve directamente favorecido por la naturaleza paralela de la GPU, los tiempos de evaluación evolucionan de forma aproximadamente constante conforme aumenta este parámetro. La diferencia entre el algoritmo con mejora y sin ella radica en un desplazamiento vertical de valores.



Como hemos visto, al aumentar la longitud de cromosoma no aumenta el paralelismo sobre la GPU. Ésa es la causa de que el tiempo de evaluación sufra el mismo patrón de crecimiento para ambos algoritmos aunque, como era de esperar, necesitándose menos esfuerzo de cómputo en el caso del algoritmo mejorado.

Coste de la Migración

Aunque la migración es siempre recomendable, resulta interesante medir el coste de la comunicación entre los dos AGs lanzados en el algoritmo paralelo (algoritmo mejorado). Esta medida también se obtiene en las pruebas automáticas.

Entendemos por coste de la comunicación el tiempo que lleva hacer las operaciones requeridas por el intercambio de individuos entre los dos procesos del AGP, y que no deberían realizarse en implementaciones sin migración. Se calcula por tanto el tiempo total que se emplea en realizar las migraciones programadas, es decir, se van acumulando los tiempos utilizados en cada una de las migraciones. Recordamos que para los resultados presentados en este capítulo se realizan 5 migraciones en total (uniformemente espaciadas), a lo largo de las generaciones CPU a completar. El tiempo de una migración individual se calcula para cada uno de los procesos, y mide:

- Para el AG sobre CPU, el tiempo transcurrido entre la emisión de la orden de migración (es la CPU la que decide cuándo se debe realizar el intercambio) y la recepción de la confirmación de que el AG sobre la GPU ha recibido los individuos (debemos esperar este evento para no corromper los individuos que se están leyendo). El proceso es el siguiente:
  1) Lanza el evento de *inicio de migración* al proceso GPU, y pasa a definir su población de intercambio.
  2) Cuando termina dicha definición lanza el evento de *listo para el intercambio*, y espera a que el algoritmo GPU le lance el mismo.
  3) A continuación recibe los individuos que le pasa la GPU, copiando la información (población intermedia vecina) en su propia población.
  4) Después lanza el evento de *fin de migración* y espera a recibir el mismo evento desde la GPU. El proceso CPU continúa después con el bucle evolutivo habitual.
- Para el AG sobre la GPU, el tiempo medido es el que transcurre entre la recepción del evento de migración (la CPU le notifica que hay que migrar) y la recepción de la confirmación de que el AG sobre la CPU ha recibido los individuos enviados. Este lapso comprende la siguiente secuencia de acciones:
  5) Define su población de intercambio. Para ello debe realizar previamente una lectura de la textura, a fin de capturar los individuos que se van a pasar al algoritmo CPU.
  6) Cuando termina dicha definición lanza el evento de *listo para el intercambio*, y espera a que el algoritmo CPU le lance el mismo.
  7) A continuación recibe los individuos que le pasa la CPU, copiando la información (población intermedia vecina) en su propia población.
  8) Después lanza el evento de *fin de migración* y espera a recibir el mismo evento desde la CPU. Finalmente, realiza una escritura en la memoria de GPU para incluir efectivamente los nuevos individuos en su propia población.

A fin de detallar el coste de comunicación, se han realizado nuevas baterías de pruebas (incluidas en el directorio \BateríasPruebas\DetallesMigración del CD-ROM) en las que se desglosan los tiempos que llevan cada una de las acciones anteriores en las que se divide la migración. Así por ejemplo, para una población de 1000 individuos, una longitud de cromosoma de 100, y 500 generaciones CPU, se obtienen los siguientes tiempos:

CPU

- Proceso de selección: 80,5316
- Proceso de recepción: 48,7406

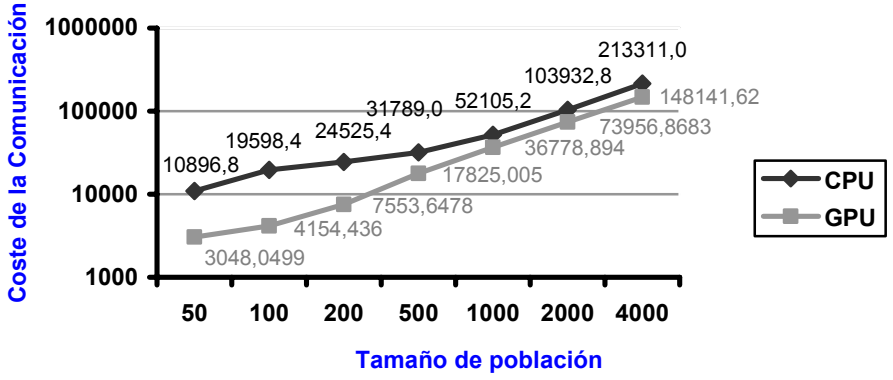
GPU

- Proceso de selección: 9,0809
- Proceso de recepción: 13,7807
- Lectura memoria GPU: 2506,75
- Escritura memoria GPU: 1002,07

Los resultados demuestran que las operaciones de acceso a la memoria de la GPU son muy costosas en tiempo, especialmente la lectura de la textura. El cronómetro asociado a la migración para el proceso CPU se detiene una vez la GPU ha recibido la población de intercambio, pero la GPU aún tiene que realizar la escritura a memoria, y su cronómetro sigue contando.

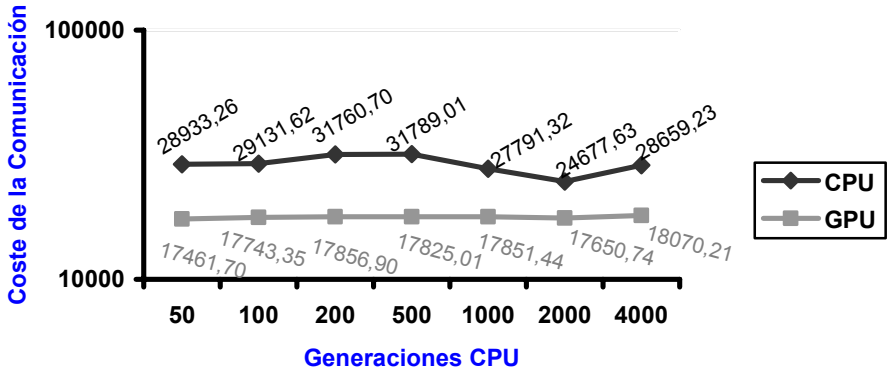
A continuación se incluyen los tiempos de comunicación obtenidos en función de los diferentes parámetros del AG. Además se ha calculado qué porcentaje supone la migración del tiempo de ejecución total (del algoritmo mejorado).

Tamaño población	Migración CPU	%Tejec	Migración GPU	%Tejec
50	10896,7913	0,9048	3048,0499	0,2531
100	19598,4173	0,9767	4154,4360	0,2070
200	24525,4149	0,6143	7553,6478	0,1892
500	31789,0102	0,3375	17825,0050	0,1893
1000	52105,2300	0,2665	36778,8940	0,1881
2000	103932,7694	0,2705	73956,8683	0,1925
4000	213310,9908	0,2614	148141,6200	0,1815

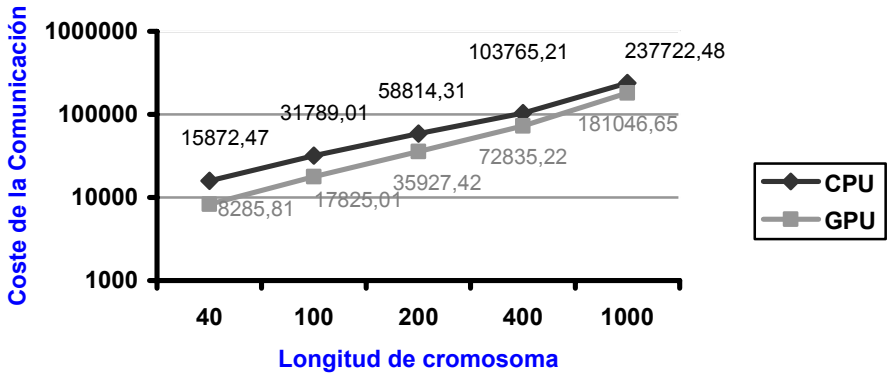


Cuanto mayor es el tamaño de la población, mayor es el coste de comunicación de los algoritmos. Esto se debe a que la población de intercambio supone aproximadamente un 4% del total, luego se intercambian más individuos cuanto mayor sea esta última. Los tiempos de ambos algoritmos se aproximan con el aumento del número de individuos, debido al crecimiento de la población de intercambio. A medida que ésta crece, los costes de selección y de recepción de la migración se acentúan para los dos algoritmos; puesto que suponen la mayor parte del tiempo de migración, la GPU debe esperar a su vecino de igual modo que la CPU.

Generaciones CPU	Migración CPU	%Tejec	Migración GPU	%Tejec
50	28933,2648	2,2432	17461,6975	1,3538
100	29131,6248	1,3472	17743,3540	0,8206
200	31760,7003	0,8312	17856,9036	0,4673
500	31789,0102	0,3375	17825,0050	0,1893
1000	27791,3222	0,1623	17851,4446	0,1042
2000	24677,6303	0,0732	17650,7403	0,0524
4000	28659,23139	0,0432	18070,21307	0,0272



Nº genes	Migración CPU	%Tejec	Migración GPU	%Tejec
40	15872,4689	0,2583	8285,8087	0,1349
100	31789,0102	0,3308	17825,0050	0,1855
200	58814,3133	0,3984	35927,4195	0,2434
400	103765,2124	0,1312	72835,2157	0,0921
1000	237722,4779	0,3468	181046,6529	0,2641



El coste de comunicación es muy similar para ambos procesos, por las razones expuestas para los resultados correspondientes a diferentes tamaños de población.

Los resultados muestran cómo los costes de comunicación son muy parecidos para ambos algoritmos. Las discrepancias existentes vienen dadas por la diferencia en el comienzo de la medición de tiempos, comentada anteriormente.

Aunque la suma de los tiempos (de selección, migración, lectura y escritura sobre la tarjeta) de la GPU es mayor que los de la CPU, el coste de migración de éste último es más alto. Esto se debe a la sincronización de intercambio de información que se realiza dentro de la fase de migración. Tanto un algoritmo como otro tienen que esperar a que su vecino termine la fase para seguir con su ejecución, por tanto, sus tiempos de migración serán parecidos. El exceso de coste registrado en la CPU se debe a que ésta comienza antes su fase de migración, y espera a que la GPU entre en ella, teniendo su cronómetro en marcha durante esta espera.

De cara a minorizar el coste de migración, parece que lo mejor es utilizar tamaños de población pequeños y longitudes de cromosoma también pequeñas. Ambos factores inciden directamente en la cantidad de información que se ha de intercambiar en cada migración (el número de individuos intercambiado es un porcentaje de la población total). Por el contrario, el número de generaciones CPU apenas influye en el coste de la comunicación debido a que el número de migraciones es siempre el mismo (en este caso, 5).

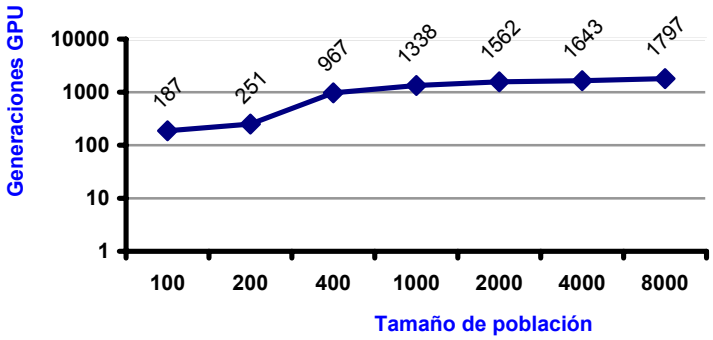
Dado que el tiempo de comunicación representa una parte muy pequeña del tiempo de ejecución total, es aconsejable guiarse por éste último a la hora de elegir los valores de los diferentes parámetros del AG.

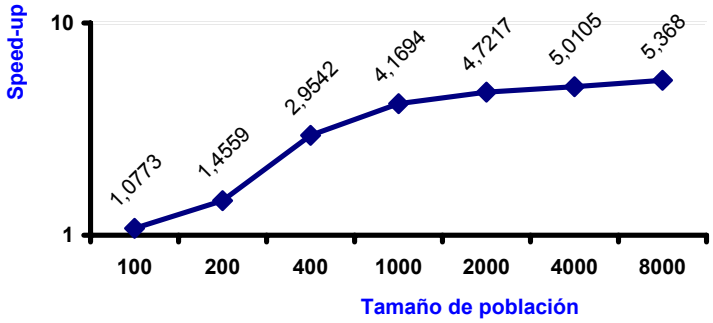
5.1.2 Función multimodal

La implementación correspondiente a la función de aptitud multimodal se caracteriza por necesitar una operación de decodificación del individuo previa a la evaluación del mismo. Veamos cómo influye esta peculiaridad en los resultados de las ejecuciones del algoritmo.

Tamaño de población

Tamaño población	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
100	187	1332156,03	1236528,01	**1,077335913**
200	251	2923106,96	2007747,27	**1,455913805**
400	967	11642949,24	3941181,26	**2,954177564**
1000	1338	39278719,45	9420673,69	**4,169417257**
2000	1562	89954544,05	19051166,89	**4,721734085**
4000	1643	192270101,04	38373514,68	**5,010489725**
8000	1797	440643111,80	82086785,72	**5,36801**

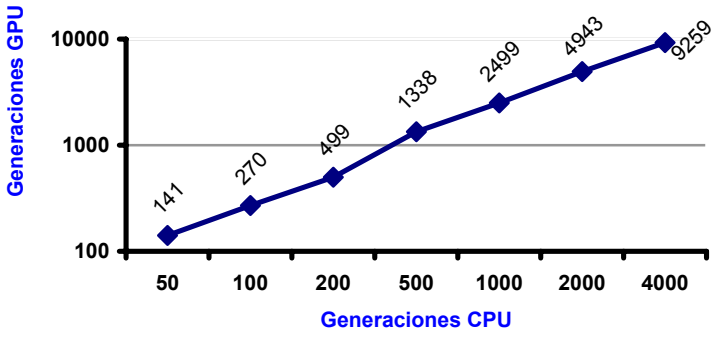


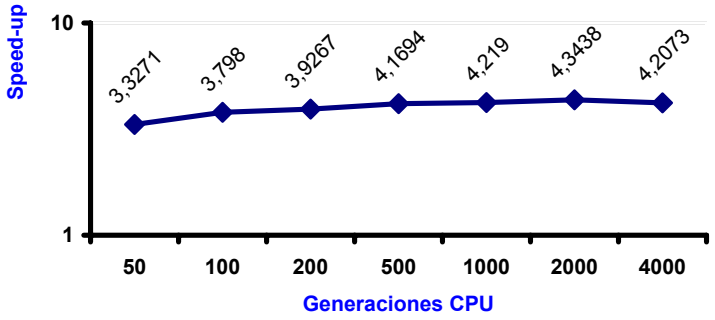


Como era de esperar, los resultados obtenidos son muy similares a los correspondientes al problema onemax, tanto en las formas de las curvas como en las cantidades alcanzadas. Se observa una ligera disminución en las magnitudes del speed-up y número de generaciones GPU obtenidos, debido a las características particulares del problema multimodal: la operación de decodificación previa a la evaluación aumenta ligeramente los cálculos a realizar para la evaluación de los individuos.

Número de generaciones

Generaciones CPU	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
50	141	4428271,96	1330983,16	**3,327068373**
100	270	8323101,67	2191466,27	**3,797960203**
200	499	15223758,01	3877029,65	**3,926655042**
500	1338	39278719,45	9420673,69	**4,169417257**
1000	2499	72593867,02	17206321,86	**4,219022963**
2000	4943	146602862,17	33750016,64	**4,343786367**
4000	9259	277647695,75	65992575,04	**4,207256583**

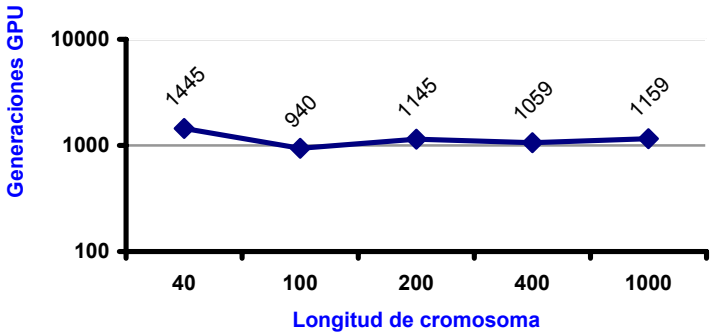


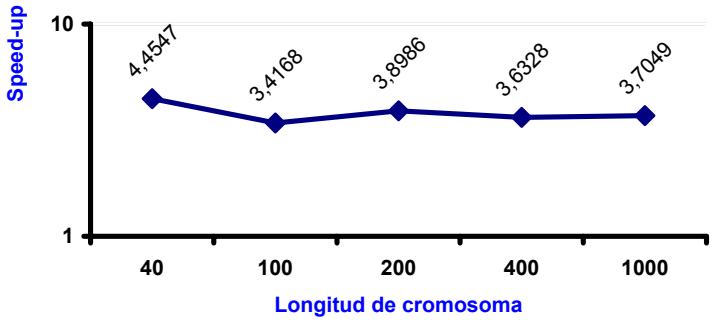


La influencia del número de generaciones CPU a realizar es análoga a la observada para el problema onemax, apareciendo de nuevo una leve disminución de los rendimientos propia del problema multimodal.

Longitud de cromosoma

Nº de genes	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
40	1445	26723144,32	5998827,87	**4,4547276**
100	940	31990497,38	9362720,73	**3,416795**
200	1145	56559769,01	14507697,20	**3,8986042**
400	1059	92188724,58	25376734,34	**3,6328049**
1000	1159	69564825,80	257733878,90	**3,70494536**



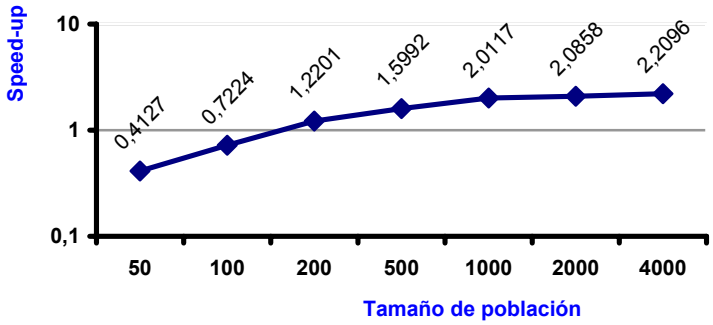


Para el parámetro de longitud de cromosoma se observa el efecto contrario que para los anteriores parámetros estudiados: se obtienen mejores resultados (generaciones GPU y speed-up) que para el problema onemax. Era de esperar que ocurriera esto dada la forma en que se realiza la operación de decodificación en cada uno de los algoritmos: mientras que en la CPU la operación de decodificación se realiza secuencialmente para cada individuo, en la GPU la inclusión de la operación de decodificación en el proceso de evaluación hace partícipe del paralelismo por columnas a la decodificación, logrando así que su impacto sobre el tiempo de ejecución sea mínimo.

A su vez se observa, en las dos gráficas correspondientes a la variación de la longitud de cromosoma, un valle para el valor por defecto (100 genes). Este fenómeno puede deberse a la forma en que se relacionan las peculiaridades del problema particular con la arquitectura interna de la GPU, para la que ciertas configuraciones pueden resultar más eficientes que otras debido a las características hardware de la tarjeta.

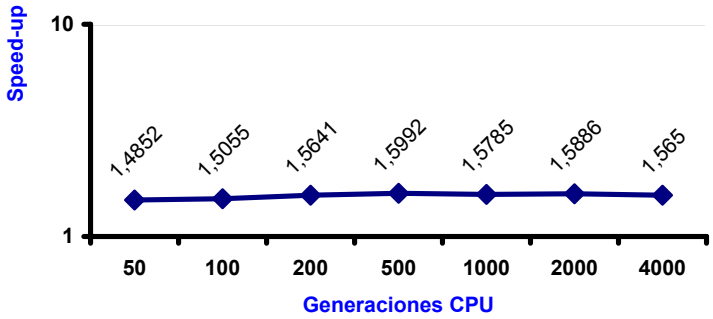
CPU vs GPU

Tamaño población	T CPU	T GPU	Speed-up
50	494497,48	1198251,14	**0,412682673**
100	964894,95	1335630,36	**0,722426637**
200	1966442,25	1611690,63	**1,220111485**
500	5203031,29	3253532,58	**1,599194465**
1000	11553774,31	5743209,13	**2,011727948**
2000	23491790,61	11262511,05	**2,085839517**
4000	50261466,14	22746921,49	**2,209594215**



Los resultados obtenidos son casi idénticos a los correspondientes para el problema anterior de onemax.

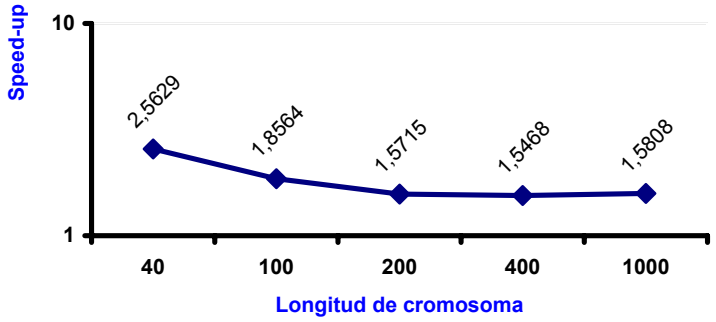
Generaciones	T CPU	T GPU	Speed-up
50	538856,60	362816,64	**1,48520366**
100	1060508,90	704404,10	**1,505540495**
200	2067821,25	1322013,39	**1,564145463**
500	5203031,29	3253532,58	**1,599194465**
1000	10142331,53	6425430,88	**1,578467144**
2000	20257201,19	12751755,23	**1,588581401**
4000	40408403,61	25820309,08	**1,564985279**



Mientras que la gráfica obtenida tiene una forma suave, recordamos que se observaba un pico pronunciado en la correspondiente a onemax (para 100 generaciones). En este caso, se mantiene dicho pico (en cuanto a la magnitud del speed-up para 100 generaciones CPU), y se obtiene un speed-up aún menor para 50 generaciones (1,48, frente a 1,69 obtenido para onemax). El bajo valor para 50 generaciones, que no se daba en el problema anterior, hace que la gráfica parezca más suave.

Para este problema parece que el algoritmo simple sobre GPU tarda más en ser más eficiente que el de CPU (que progresa de forma parecida para todos los problemas).

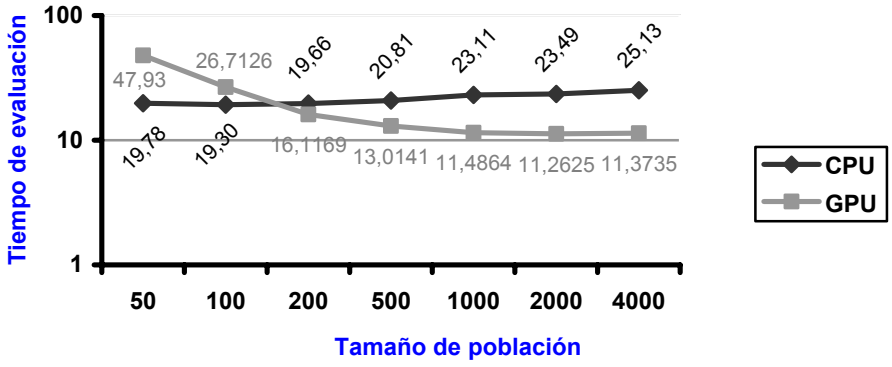
Nº genes	T CPU	T GPU	Speed-up
40	3797560,68	1481753,69	**2,5628826**
100	5995513,64	3229616,64	**1,8564165**
200	9725588,93	6188649,50	**1,5715204**
400	17836929,57	11531401,35	**1,5468137**
1000	47655062,67	30146123,31	**1,58080235**

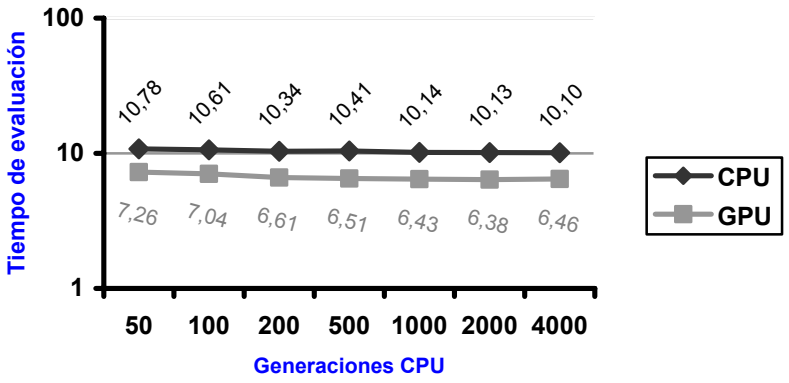


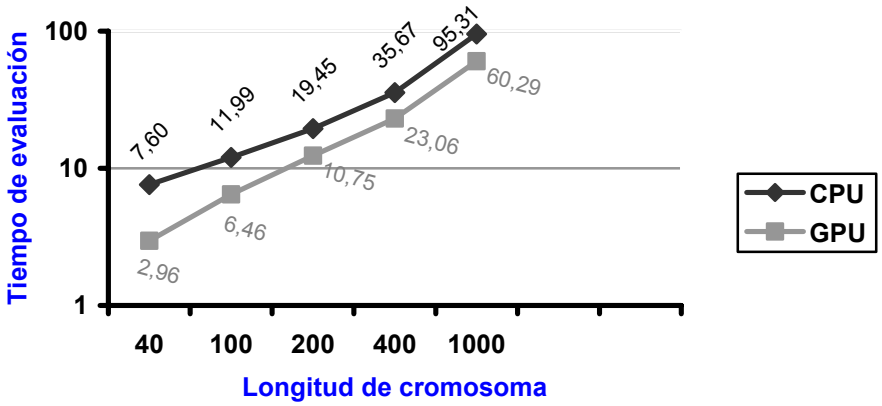
El comportamiento para diferentes longitudes de cromosoma es equivalente al observado para onemax, resultando el algoritmo GPU ligeramente más eficiente que el CPU en este caso.

Esfuerzo computacional

La incorporación de la operación adicional de decodificación en el proceso de evaluación de los individuos provoca un leve incremento en el tiempo de evaluación, y por tanto en el esfuerzo computacional que requiere el AG.



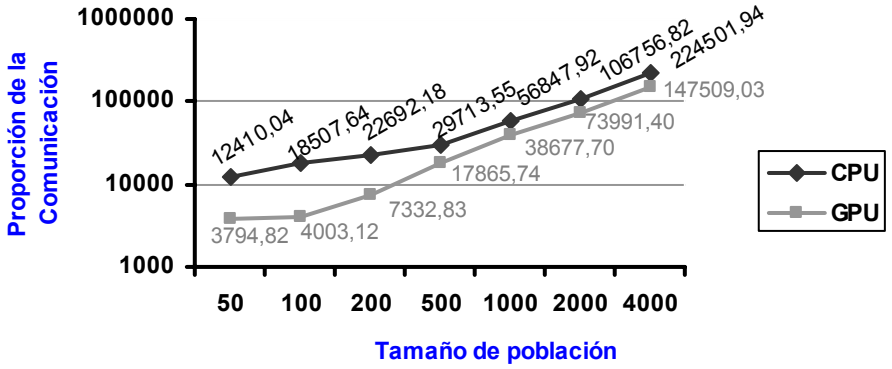




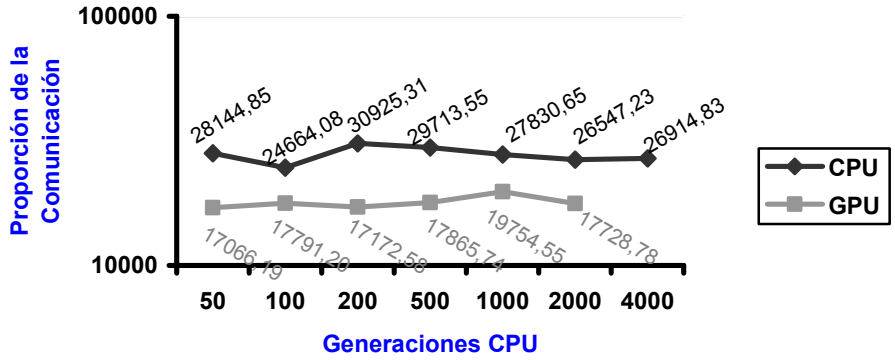
Coste de la Migración

El coste de la migración para la CPU es parecido al obtenido para el problema onemax.

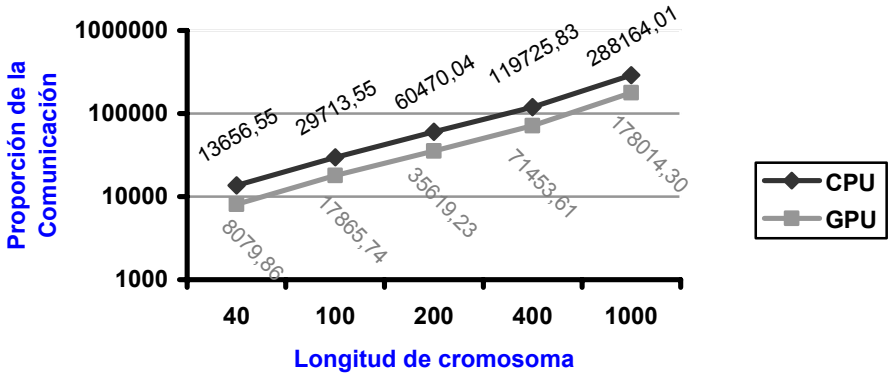
Tamaño población	Migración CPU	%Tejec	Migración GPU	%Tejec
50	12410,0357	1,0036	3794,8239	0,3069
100	18507,6444	0,9218	4003,1171	0,1994
200	22692,1793	0,5758	7332,8301	0,1861
500	29713,5504	0,3154	17865,7435	0,1896
1000	56847,9182	0,2984	38677,7041	0,2030
2000	106756,8242	0,2782	73991,3971	0,1928
4000	224501,9416	0,2735	147509,0316	0,1797



Generaciones CPU	Migración CPU	%Tejec	Migración GPU	%Tejec
50	28144,8528	2,1146	17066,1895	1,2822
100	24664,0806	1,1255	17791,2039	0,8118
200	30925,3074	0,7977	17172,5840	0,4429
500	29713,5504	0,3154	17865,7435	0,1896
1000	27830,6463	0,1617	19754,5523	0,1148
2000	26547,2277	0,0787	17728,7807	0,0525
4000	26914,8318	0,0408	17751,4117	0,0269



Nº genes	Migración CPU	%Tejec	Migración GPU	%Tejec
40	13656,5501	0,2277	8079,8606	0,1347
100	29713,5504	0,3174	17865,7435	0,1908
200	60470,0395	0,4168	35619,2299	0,2455
400	119725,8321	0,4718	71453,6149	0,2816
1000	288164,0121	0,1118	178014,2965	0,0691

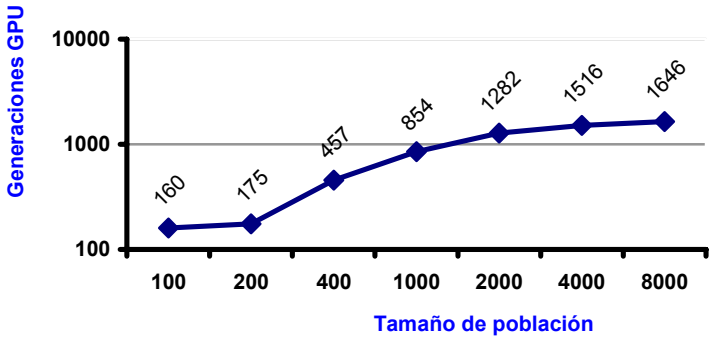


5.1.3 Función defectiva

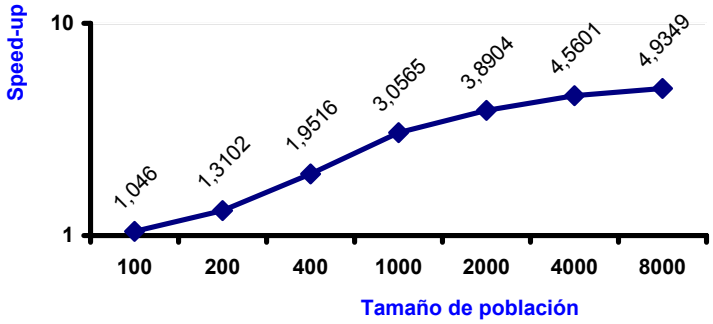
Como sabemos, la mayor parte del tiempo de ejecución de un AG se debe a la evaluación de los individuos, y ésta a su vez es particular para cada problema. En el caso defectivo, la evaluación se realiza por bloques, de modo que dentro de un bloque se sigue un proceso similar a la evaluación realizada en el problema onemax (ha de contarse el número de genes con valor 1) y, cada vez que se completa un bloque, se aplica la función defectiva para calcular la aportación de dicho bloque al fitness del individuo. Es de esperar que los tiempos de evaluación se vean incrementados con respecto a los obtenidos para el problema onemax.

Tamaño de población

Tamaño población	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
100	160	1281458,18	1225129,16	**1,045978021**
200	175	2629867,90	2007238,82	**1,310191829**
400	457	7734903,16	3963398,37	**1,951583576**
1000	854	29574447,11	9676010,46	**3,05647118**
2000	1282	78951079,63	20293925,64	**3,890379863**
4000	1516	184103032,43	40372638,91	**4,560094099**
8000	1646	417460233,86	84593185,83	**4,934915617**



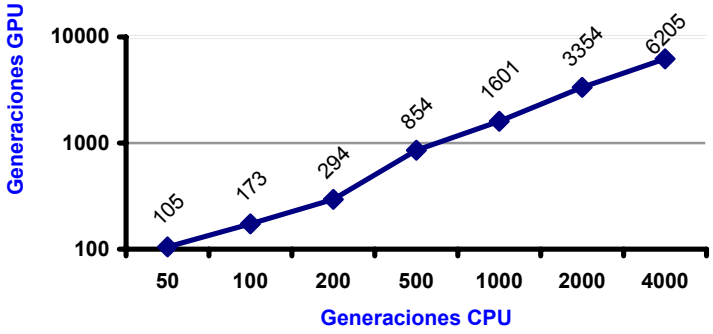
Para el problema defectivo, el número de generaciones que realiza el proceso GPU presenta un valle inicial más extenso que en los problemas anteriores, para los que se alcanzaban alrededor de 1400 generaciones GPU con poblaciones de 1000 individuos. La forma de la gráfica se aproxima más a un crecimiento lineal para este problema.



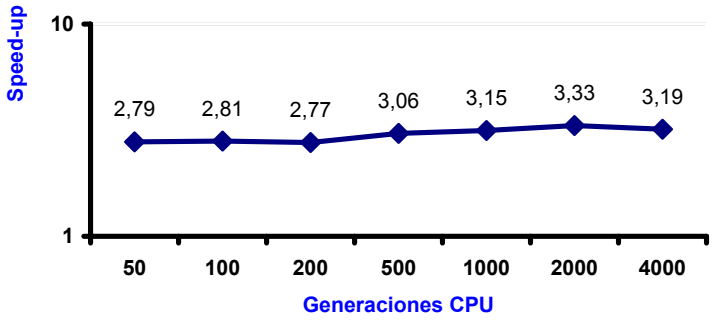
El speed-up también se aproxima más a un crecimiento lineal que en los anteriores problemas.

Número de generaciones

Generaciones CPU	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
50	105	3733098,54	1340005,30	**2,785883404**
100	173	6278982,63	2234711,94	**2,809750333**
200	294	11012099,81	3982067,46	**2,765422715**
500	854	29574447,11	9676010,46	**3,05647118**
1000	1601	55478998,84	17609203,11	**3,150568398**
2000	3354	114515286,22	34427643,87	**3,326259754**
4000	6205	215503523,11	67480575,09	**3,193563819**



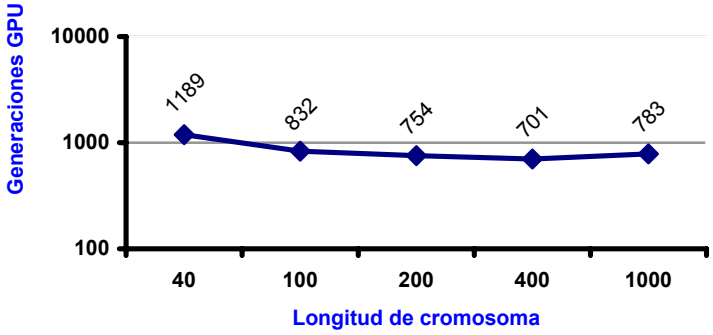
La forma aproximadamente lineal se corresponde con la obtenida para los problemas anteriores, aunque los valores son menores. La curva está menos acentuada en el crecimiento debido a que, como se observa en la siguiente gráfica, la mejora aportada por el algoritmo paralelo no es tan grande como en los casos anteriores.

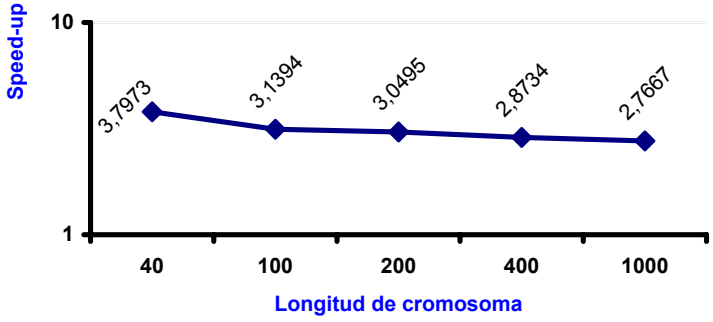


Longitud de cromosoma

Para este parámetro, los resultados son similares a los conseguidos en los problemas anteriores.

Nº de genes	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
40	1189	6014493,03	22838873,36	**3,7973065**
100	832	29517371,55	9402107,91	**3,1394419**
200	754	43820581,51	14369851,42	**3,0494805**
400	701	25413098,00	73020774,60	**2,8733519**
1000	783	191205567,92	69108834,42	**2,76673119**

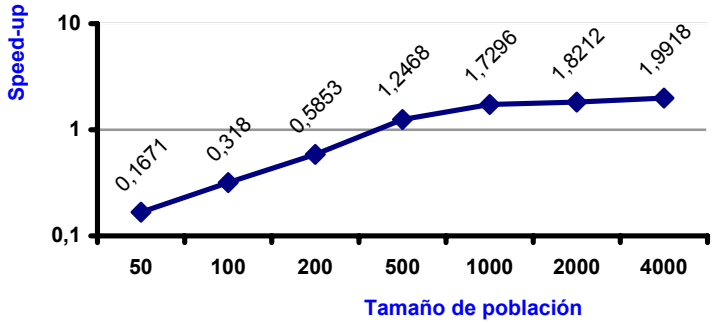




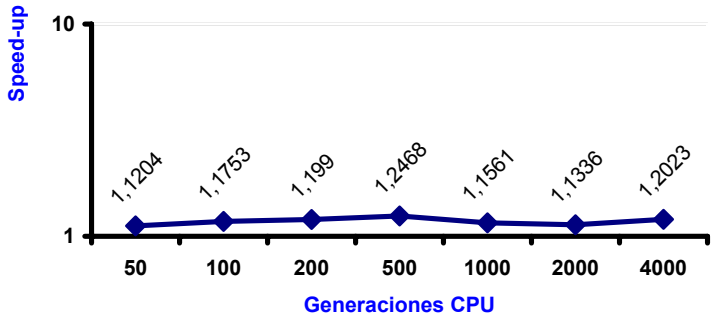
CPU vs GPU

En los resultados que comparan ambas implementaciones simples (CPU y GPU) se refleja más claramente lo que se intuía en los resultados anteriores: para el problema defectivo, el algoritmo que usa la GPU no es tan eficiente con respecto al algoritmo puramente CPU.

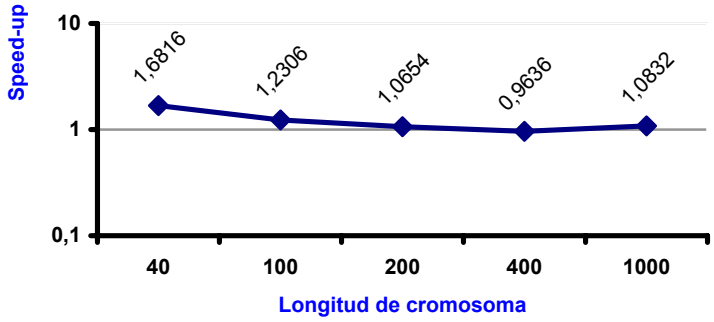
Tamaño población	T CPU	T GPU	Speed-up
50	479798,15	2871065,20	**0,167115032**
100	969399,32	3048660,68	**0,317975474**
200	1965453,42	3357825,01	**0,585335273**
500	5265908,55	4223598,34	**1,246782512**
1000	11624832,78	6721303,18	**1,729550427**
2000	23646375,78	12984091,78	**1,821180578**
4000	50273839,94	25240343,39	**1,991804912**



Generaciones	T CPU	T GPU	Speed-up
50	579209,13	516948,84	**1,120438002**
100	1095552,12	932116,10	**1,175338693**
200	2114750,90	1763733,15	**1,199019764**
500	5265908,55	4223598,34	**1,246782512**
1000	10263242,40	8877360,71	**1,156114158**
2000	20463772,32	18051812,57	**1,133613162**
4000	40472363,50	33662597,26	**1,202294736**

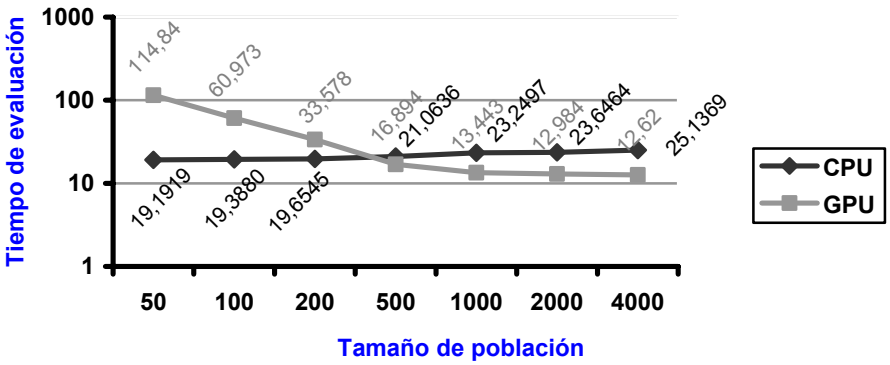


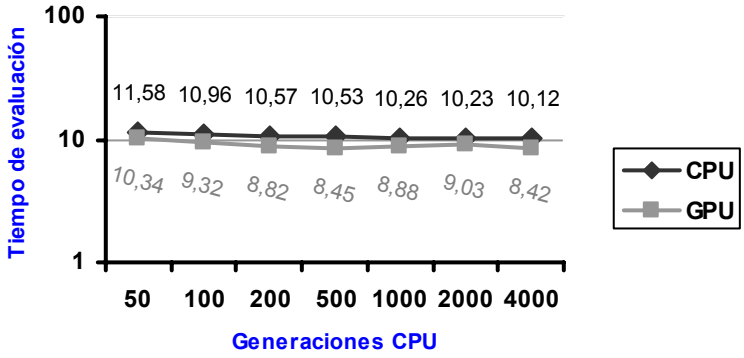
Nº genes	T CPU	T GPU	Speed-up
40	3406273,69	2025585,70	**1,6816241**
100	5200729,36	4226295,80	**1,2305644**
200	8150526,60	7650471,44	**1,0653627**
400	14622209,58	15175203,97	**0,9635593**
1000	39993242,72	36922082,79	**1,08317949**

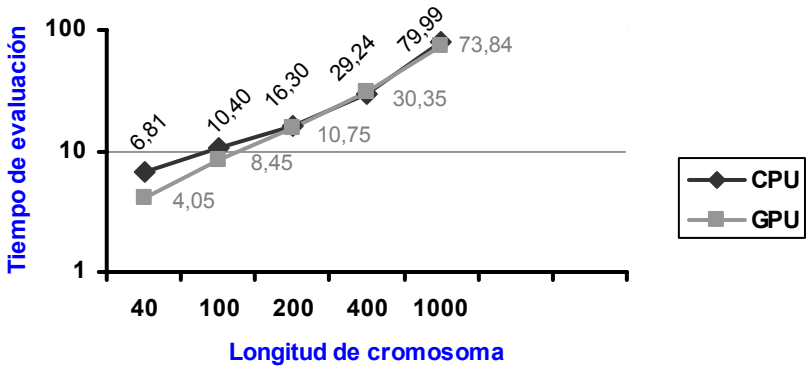


Esfuerzo computacional

De nuevo queda patente cómo el algoritmo GPU no obtiene tan buenos resultados para el problema defectivo como para los vistos anteriormente, acortándose así la distancia con el algoritmo secuencial CPU.



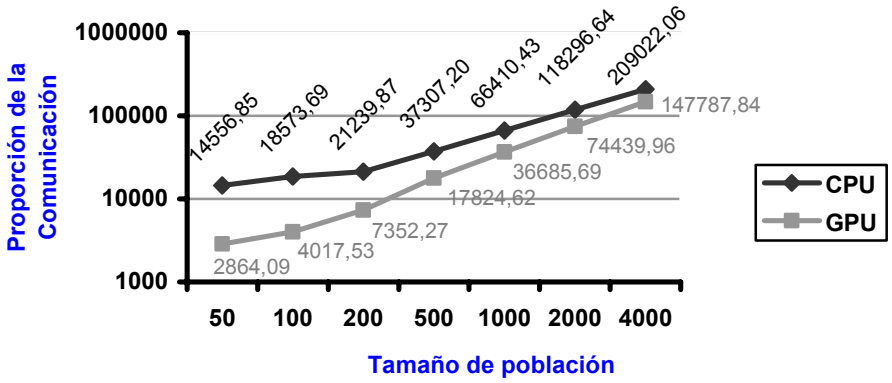




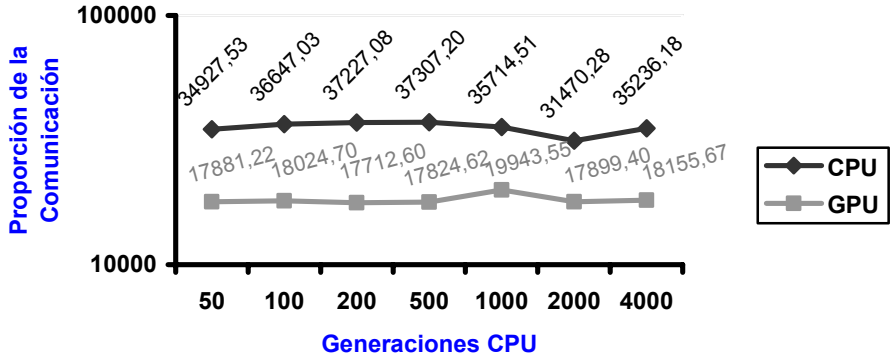
Coste de la Migración

Los tiempos de comunicación de la implementación para el problema defectivo son muy similares a los obtenidos para la función de aptitud multimodal.

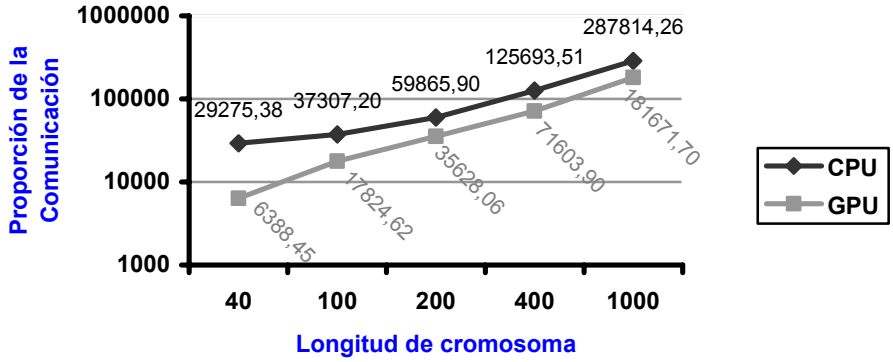
Tamaño población	Migración CPU	%Tejec	Migración GPU	%Tejec
50	14556,852	1,1882	2864,0893	0,2338
100	18573,691	0,9253	4017,5251	0,2002
200	21239,87	0,5359	7352,2675	0,1855
500	37307,2	0,3856	17824,622	0,1842
1000	66410,4323	0,3272	36685,6857	0,1808
2000	118296,64	0,2930	74439,961	0,1844
4000	209022,06	0,2471	147787,84	0,1747



Generaciones CPU	Migración CPU	%Tejec	Migración GPU	%Tejec
50	34927,533	2,6065	17881,218	1,3344
100	36647,032	1,6399	18024,696	0,8066
200	37227,084	0,9349	17712,601	0,4448
500	37307,2	0,3856	17824,622	0,1842
1000	35714,5124	0,2028	19943,5532	0,1133
2000	31470,277	0,0914	17899,396	0,0520
4000	35236,18	0,0522	18155,67	0,0269



Nº genes	Migración CPU	%Tejec	Migración GPU	%Tejec
40	29275,382	0,1282	6388,4504	0,0280
100	37307,2	0,3968	17824,622	0,1896
200	59865,895	0,4166	35628,06	0,2479
400	125693,51	0,1721	71603,895	0,0981
1000	287814,256	0,4165	181671,701	0,2629



5.1.4 Solución al problema defectivo

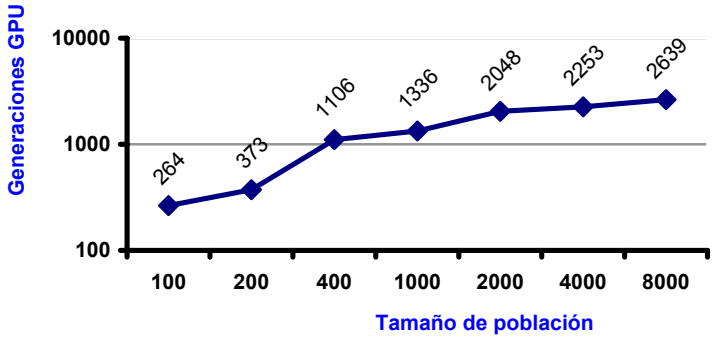
La implementación del algoritmo de mapeo evolutivo (explicado en el apartado 4.5 del capítulo de implementación) añade un cromosoma de mapeo en los individuos de la población. La estructura de datos del individuo se verá por tanto incrementada al albergar el doble de información (el genotipo y el cromosoma de mapeo). Además, la evaluación requiere de una operación previa en la que, mapeando el genotipo del individuo, se construye el fenotipo a evaluar.

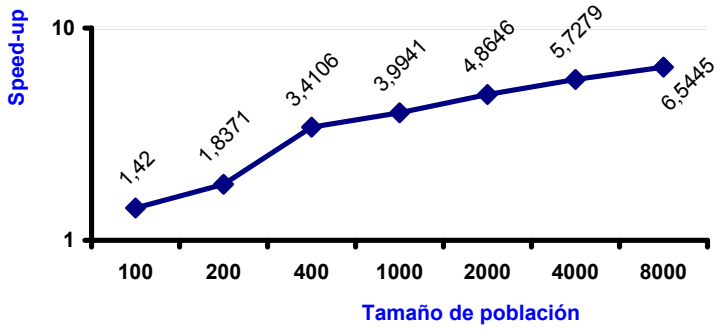
Tamaño de población

La construcción del fenotipo, en el algoritmo GPU, se realiza paralelamente para todos los individuos (paralelismo por columnas). En el algoritmo CPU, por el contrario, supone una operación secuencial adicional en el cálculo de la evaluación de cada individuo. Por tanto, el algoritmo paralelo goza de mayor ventaja (gracias al paralelismo

de la GPU) frente al algoritmo secuencial, que en la aproximación tradicional del problema defectivo (sin cromosomas de mapeo).

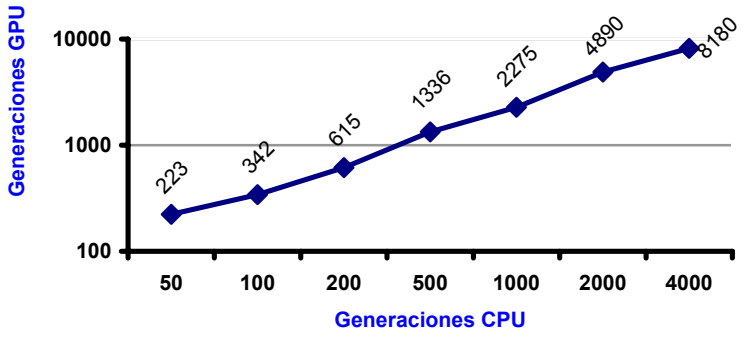
Tamaño población	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
100	264	3082732,89	2170975,29	**1,419976**
200	373	7069333,60	3848188,16	**1,8370551**
400	1106	28562101,20	8374521,33	**3,41059507**
1000	1336	89647643,80	22445150,69	**3,99407627**
2000	2048	245907170,77	50549824,86	**4,86464931**
4000	2253	561279503,94	97990614,16	**5,72789046**
8000	2639	1497657527,96	228841071,07	**6,544531193**



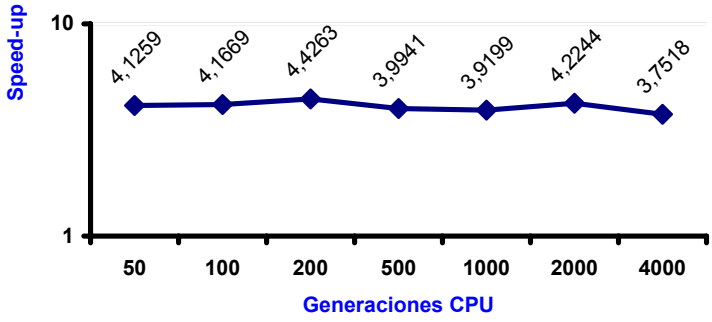


Número de generaciones

Generaciones CPU	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
50	223	15016789,36	3639661,89	**4,1258748**
100	342	23225432,04	5573739,76	**4,1669387**
200	615	40665492,46	9187142,69	**4,42634819**
500	1336	89647643,80	22445150,69	**3,99407627**
1000	2275	149937005,38	38250155,95	**3,91990573**
2000	4890	309537031,08	73272881,82	**4,22444189**
4000	8180	545731678,14	145457547,06	**3,751827864**



Gracias a la mejora del rendimiento que el paralelismo produce en el algoritmo GPU, el número de generaciones GPU que se logran es mucho mayor que el correspondiente a la implementación del problema defectivo sin mejora. En consecuencia, el speed-up también es considerablemente mayor, aunque tiende a disminuir conforme crece el número de generaciones CPU (al contrario de lo que ocurría en la aproximación tradicional).

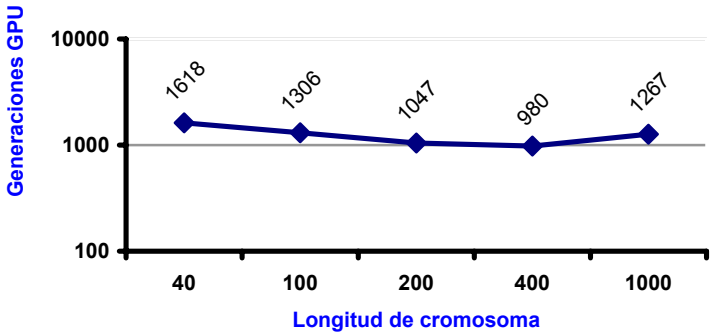


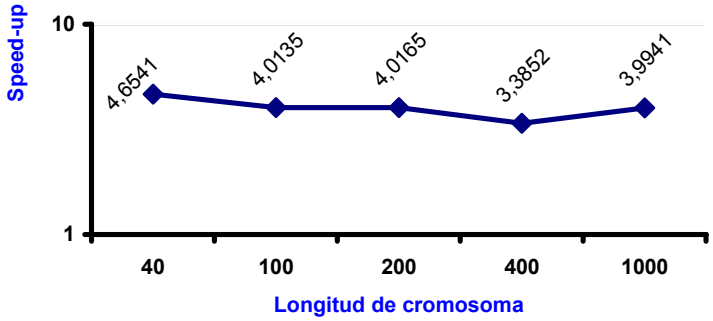
Longitud de cromosoma

Puesto que para esta implementación los individuos constan de dos cromosomas (genes y mapeo), la zona de datos de los individuos ocupa el doble que en la implementación sin mapeo evolutivo. En la CPU, la operación de construcción del fenotipo exige recorrer secuencialmente el cromosoma de mapeo de cada individuo, y por otro lado la función defectiva se aplica sobre el fenotipo construido, cuya longitud es igual a la del cromosoma de mapeo (y el de genes). El proceso de evaluación global, por tanto, supone un recorrido secuencial de $2 \cdot lcrom$ elementos, es decir, del doble de la longitud de cromosoma considerada en la implementación tradicional. Por el contrario, en la GPU, la construcción del fenotipo (a partir del cromosoma de mapeo) se realiza totalmente en paralelo, aunque la aplicación posterior de la función de fitness se mantiene secuencial en los genes (paralelismo sólo por columnas).

Además de repetirse una mayor eficiencia del algoritmo paralelo con mapeo evolutivo con respecto al algoritmo paralelo sin esta mejora, la forma en que el crecimiento de la longitud del cromosoma influye sobre el speed-up y las generaciones GPU no es tan marcada. Así, se observa que la pérdida de eficiencia conforme crece la longitud del cromosoma no es tan rápida.

Nº de genes	Generaciones GPU	T sin Mejora	T con Mejora	Speed-up
40	1618	62910110,36	13517233,75	**4,6540669**
100	1306	88944796,62	22161526,07	**4,0134780**
200	1047	134130377,67	33394489,91	**4,01654219**
400	980	224758591,24	66394386,47	**3,38520473**
1000	1267	730958745,69	183007472,70	**3,99414699**

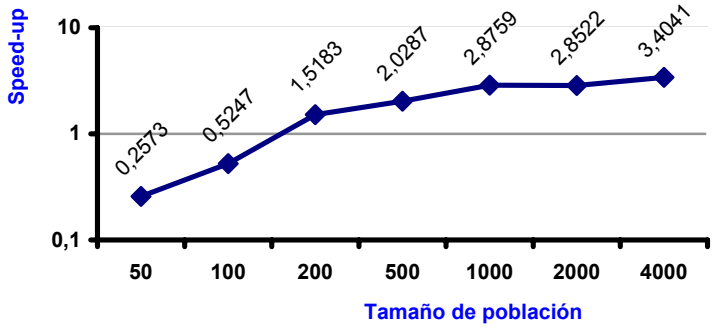




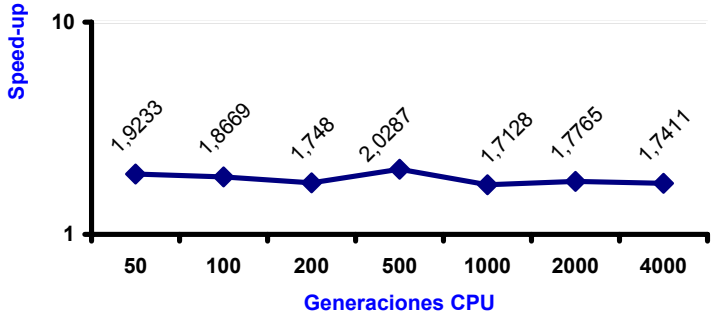
CPU vs GPU

Los resultados de este apartado confirman que el algoritmo simple GPU es mucho más eficiente que el algoritmo CPU, en relación a la comparativa realizada para la implementación sin mapeo evolutivo. Los tiempos de ejecución son aquí mayores para ambos algoritmos, respecto a los obtenidos para la implementación tradicional del problema defectivo. Debido al mayor volumen de datos a procesar y a la operación adicional de construcción del fenotipo, la distancia entre estos tiempos es mayor: por ejemplo, obtenemos un speed-up del 3,4 frente al 1,2 obtenido sin mapeo evolutivo (para 4000 individuos, cromosomas de longitud 100 y 500 generaciones CPU).

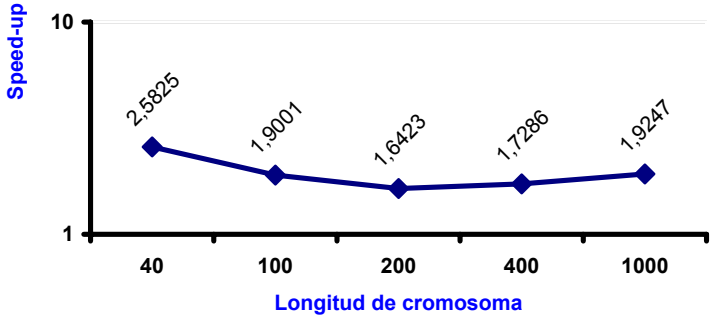
Tamaño población	T CPU	T GPU	Speed-up
50	1095347,38	4256975,07	**0,2573065**
100	2152914,94	4102818,42	**0,5247405**
200	4662088,89	3070504,44	**1,51834625**
500	12971773,13	6394272,47	**2,02865505**
1000	32227592,24	11206112,86	**2,87589396**
2000	62529454,77	21923411,64	**2,8521772**
4000	148209773,83	43538706,33	**3,404092274**



Generaciones	T CPU	T GPU	Speed-up
50	1366996,06	710737,87	**1,9233477**
100	2631587,28	1409621,52	**1,8668751**
200	4971940,47	2844311,96	**1,74802924**
500	12971773,13	6394272,47	**2,02865505**
1000	23787374,03	13888159,34	**1,7127809**
2000	47882045,68	26953272,81	**1,77648355**
4000	93568710,67	53742111,54	**1,741068744**

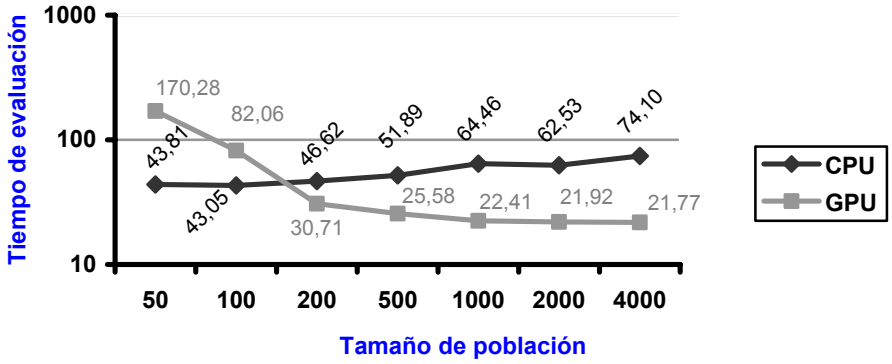


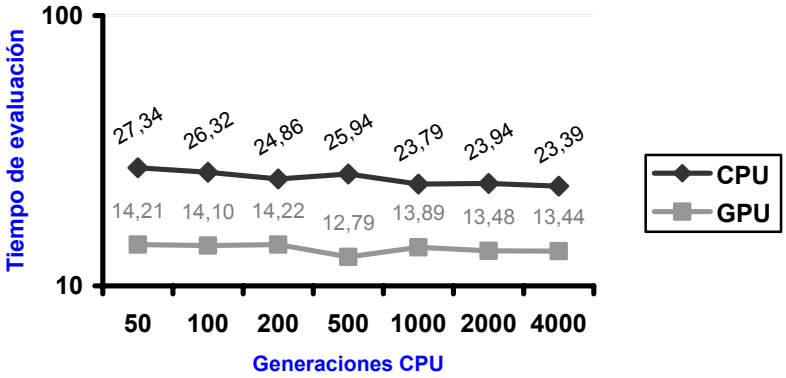
Nº genes	T CPU	T GPU	Speed-up
40	8017435,70	3104469,24	**2,5825463**
100	12923443,23	6801361,15	**1,900126**
200	21517646,52	13102534,57	**1,6422507**
400	41529466,43	24024691,14	**1,72861604**
1000	121827622,44	63296606,53	**1,92471017**

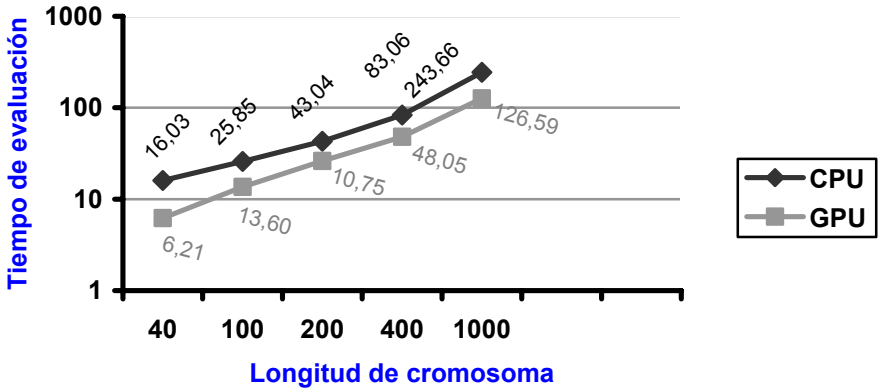


Esfuerzo computacional

Al contar con una evaluación más compleja que la aproximación tradicional al problema defectivo, el esfuerzo computacional que requiere el mapeo evolutivo es significativamente mayor. Sin embargo, gracias al paralelismo total aplicado a la construcción del fenotipo, la diferencia entre los tiempos de evaluación de los algoritmos simples (CPU y GPU) es mayor que la obtenida para la implementación sin esta mejora.



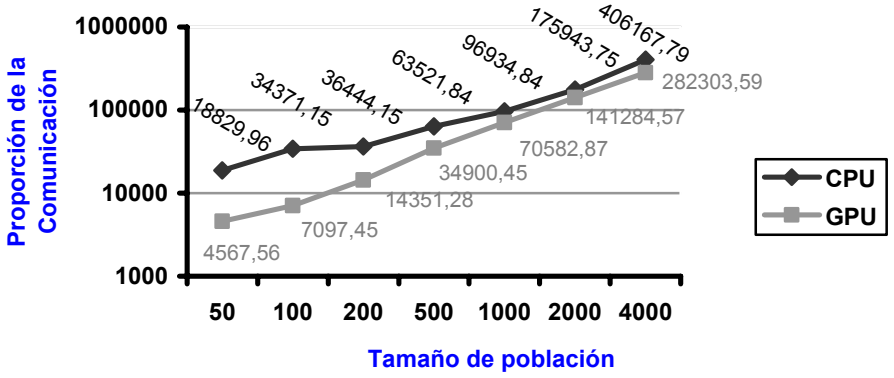




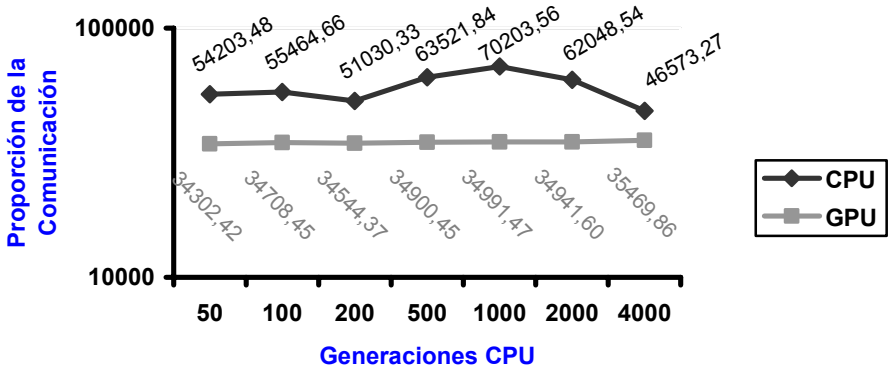
Coste de la Migración

El aumento del volumen de información que almacena un individuo produce el aumento proporcional del volumen de datos que se intercambia durante la migración. Así pues, los tiempos de migración son mayores que en el caso de individuos con un único cromosoma (genotipo). Al aumentar, por otro lado, el tiempo de ejecución total del algoritmo paralelo, las porciones del tiempo total debidas a la migración son muy similares a las del algoritmo sin mapeo.

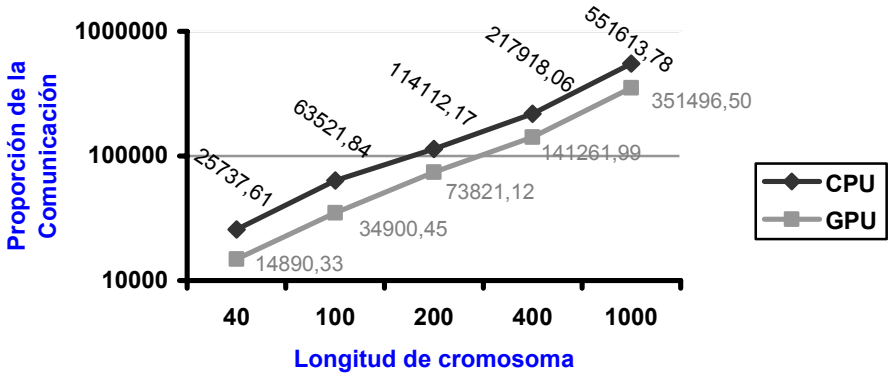
Tamaño población	Migración CPU	%Tejec	Migración GPU	%Tejec
50	18829,9572	0,8674	4567,55799	0,2104
100	34371,1532	0,8932	7097,45061	0,1844
200	36444,14658	0,4352	14351,28149	0,1714
500	63521,83518	0,2830	34900,4479	0,1555
1000	96934,84278	0,1918	70582,86519	0,1396
2000	175943,75	0,1796	141284,57	0,1442
4000	406167,79	0,1775	282303,59	0,1234



Generaciones CPU	Migración CPU	%Tejec	Migración GPU	%Tejec
50	54203,483	1,4892	34302,421	0,9425
100	55464,663	0,9951	34708,446	0,6227
200	51030,3327	0,5555	34544,3691	0,3760
500	63521,8352	0,2830	34900,4479	0,1555
1000	70203,5614	0,1835	34991,4676	0,0915
2000	62048,54	0,0847	34941,6	0,0477
4000	46573,27	0,0320	35469,86	0,0244



Nº genes	Migración CPU	%Tejec	Migración GPU	%Tejec
40	25737,615	0,1904	14890,332	0,1102
100	63521,835	0,2866	34900,448	0,1575
200	114112,168	0,3417	73821,1218	0,2211
400	217918,061	0,3282	141261,993	0,2128
1000	551613,776	0,3014	351496,499	0,1921



5.2 Conclusiones

5.2.1 Onemax

A la luz de los datos del apartado anterior, vemos que el uso de la GPU proporciona una gran ganancia en tiempo de ejecución: hasta un 5,4 de speed-up. Analizando la forma en que cada uno de los parámetros del AG influye en el rendimiento obtenido, podemos realizar las siguientes recomendaciones:

- Cuanto mayor sea el tamaño de la población, mayor rendimiento obtendremos. De hecho éste es el factor con mayor peso a la hora de mejorar el rendimiento, puesto que es el que afecta directamente en el grado de paralelismo que se logra en la GPU. Sin embargo, hay que tener en cuenta que el tiempo es a menudo un recurso limitado. Aunque a menor ritmo que en la versión no mejorada, los tiempos de ejecución para el algoritmo que usa la GPU también crecen con el número de individuos. El usuario deberá establecer, en último término, un compromiso entre la mejora de rendimiento que desea obtener y el tiempo de ejecución que está dispuesto a invertir. Se observa además que a partir de 1000 individuos el aumento en el rendimiento se estanca.

  Como dato orientativo adicional, cabe señalar que el tiempo de evaluación correspondiente a la versión mejorada se estabiliza a partir de 500 individuos (tomando valores por debajo de 12µs). Es decir, a partir de un tamaño de 500 el esfuerzo de cálculo se mueve cerca del mínimo, estableciendo así el punto a partir del cual el algoritmo mejorado comienza a ser eficiente por sí mismo (no en comparación con la versión CPU).

- El siguiente factor en importancia resulta ser la longitud de cromosoma y, puesto que su crecimiento no aumenta el grado de paralelismo en la parte ejecutada sobre la GPU, es recomendable usar el mínimo número de genes posible. El tiempo de evaluación crece proporcionalmente con este parámetro, de modo que es recomendable utilizar una longitud máxima de 400 genes (obtendremos así un speed-up mayor de 3).

- El número de generaciones a realizarse en la CPU es el factor menos influyente en la mejora del rendimiento, por tanto la elección del valor de dicho parámetro puede realizarse en función de la calidad de las soluciones deseada (en vez de en función del speed-up). Según el speed-up y el tiempo de evaluación, a partir de 200 generaciones se obtienen valores muy similares.

En resumen, es recomendable usar tamaños de población elevados (al menos hasta alcanzar los 1000 individuos) y cromosomas de poca longitud (como máximo de 400 genes).

5.2.2 Función multimodal

Las medidas de eficiencia obtenidas para este problema son muy parecidas a las del onemax. A la vista de los resultados experimentales, parece que las indicaciones realizadas en el apartado anterior se reafirman.

5.2.3 Función defectiva

Aunque la forma en que evoluciona el speed-up en función de los diferentes parámetros del AG es similar a la obtenida para los problemas anteriores, se observa una atenuación en cuanto a la magnitud de la influencia de dichos parámetros. Es decir, tarda más en alcanzar un buen speed-up, y además el speed-up conseguido no es tan bueno como para el resto de implementaciones. Estas peculiaridades se deben a las características del problema defectivo, es decir, a las operaciones requeridas por la evaluación de los individuos:

- El factor de mayor peso es de nuevo el tamaño de la población, de modo que cuantos más individuos se mantengan, mayor será el rendimiento obtenido. Para este problema el crecimiento del speed-up es más lineal y, mientras que para 1000 individuos obteníamos un speed-up de 4,3 en onemax, ahora no superamos este rendimiento hasta alcanzar un tamaño de población de 4000 individuos.
- El siguiente factor en importancia también es la longitud de cromosoma, e influye de modo similar al resto de problemas.
- El número de generaciones a realizarse en la CPU es de nuevo el factor menos influyente en la mejora del rendimiento. A partir de 500 generaciones CPU, se obtienen valores de speed-up superiores a 3.

Por tanto, se recomienda de nuevo utilizar tamaños de población elevados y longitud de cromosomas pequeñas. Además, en este problema resulta mayor la influencia del tamaño de población sobre el speed-up que en los anteriores, y por tanto debe intentarse con especial interés manejar el mayor número de individuos posibles. Para los problemas anteriores, sin embargo, se alcanzaba un umbral a partir del cual el aumento del valor de este parámetro no ofrecía un aumento significativo del rendimiento, y sí podía provocar un aumento excesivo en el consumo de recursos. Para el caso del problema defectivo, y para los tamaños de población considerados, no hemos alcanzado este umbral que define el comienzo de la estabilidad del speed-up.

5.2.4 Solución al problema defectivo

El speed-up obtenido para la implementación del algoritmo de mapeo evolutivo es el mayor de todos. Recordamos que, en contraste, los speed-up obtenidos para la implementación del problema defectivo (sin solucionar) son los menores. La operación de construcción del fenotipo a partir de los cromosomas de datos (genotipo y mapeo) resulta por tanto mucho más costosa para el algoritmo sobre la CPU que para el algoritmo que usa la GPU, gracias al paralelismo que ofrece el procesador de la tarjeta gráfica.

Cuanto mayor es el tamaño de la población, mayor rendimiento obtenemos, llegándose a alcanzar un speed-up de más de 6,5. En la medida de lo posible, conviene reducir la longitud de cromosoma, aunque en este caso el deterioro del rendimiento es menos acentuado (se obtiene un speed-up de 4,65 para una longitud de cromosoma de 40, frente a un 3,99 para 1000 genes).

6 Conclusiones

6.1 Objetivos Cumplidos

Los algoritmos genéticos hacen evolucionar una población de soluciones a lo largo de un número de generaciones. En este proceso, cada individuo es independiente de cualquier otro, y su evolución se produce de forma paralela al del resto de individuos.

Las tarjetas gráficas programables nos permiten ejecutar código que lleve a cabo los cálculos implicados en el proceso de evolución de la población. Los resultados experimentales obtenidos en este proyecto constituyen una prueba de que resulta factible dividir un AG secuencial en pequeños kernels que realizan las operaciones genéticas sobre los individuos de una población, mantenida en la memoria interna de la GPU. De este modo se usa la potencia desaprovechada de este dispositivo hardware.

Debido a la alta capacidad de cómputo paralelo característica de las tarjetas gráficas, éstas parecen una buena plataforma para la evolución de una población de individuos. Hemos podido constatar, para los problemas abordados en este trabajo, que la versión de un AG simple que utiliza la GPU se ejecuta hasta 3,4 veces más rápido que la versión tradicional del AG que sólo utiliza la CPU[1].

Aprovechando los dos procesadores disponibles en el sistema (CPU y procesador de fragmentos de la GPU) para implementar un AG paralelo de grano grueso, en el que cada procesador mantiene una población de individuos que migran periódicamente entre sí, era de esperar obtener una mejora en el tiempo de ejecución respecto al AG simple ejecutado únicamente sobre el procesador CPU. Añadiendo a esta mejora el hecho de que el procesador de la tarjeta gráfica trabaja de forma paralela, y que esta característica lo hace especialmente indicado para trabajar con AGs, el speed-up esperado sería aún mayor. Los resultados obtenidos han confirmado las expectativas iniciales, de modo que el tiempo de ejecución del AG paralelizado sobre la CPU y la GPU llega a ser cerca de 6,5 veces menor que el correspondiente al AG simple sobre la CPU.

Algunos de los problemas que resultan *duros* para los AGs son los problemas defectivos. Hemos comprobado que estos algoritmos también pueden ser implementados sobre la tarjeta gráfica, y que además, obtienen una mejora de eficiencia similar a la del resto de problemas considerados. Como objetivo adicional, surgido durante el proceso de desarrollo y realización de pruebas, nos planteamos la implementación de un algoritmo capaz de obtener mejores soluciones para los

[1] *Para un tamaño de población de 4000 individuos, una longitud de cromosoma de 100, y 500 generaciones.*

problemas defectivos. El algoritmo de mapeo evolutivo propuesto en [19], y cuya mejora en cuanto a la calidad de las soluciones obtenidas hemos dado por supuesta, se ha implementado a fin de comprobar si nuestra propuesta de paralelización era también válida. Los resultados en cuanto a la mejora en tiempo de ejecución han sido mucho mejores que los correspondientes a la aproximación tradicional de los problemas defectivos. Añadiendo estos resultados a la mejora de fitness, el mapeo evolutivo se revela como la aproximación más adecuada para el problema defectivo en nuestra propuesta de AG paralelo utilizando la GPU.

Una vez realizado el estudio de la influencia de los diferentes parámetros del AG sobre el speed-up, para sacar el máximo rendimiento a nuestro modelo de paralelización, conviene, en la medida de lo posible, maximizar la relación tamaño de población-longitud de cromosoma. Dado que, la paralelización implementada sobre la GPU para la evaluación se realiza por columnas, el tamaño de población es el factor más influyente en la mejora de la eficiencia del modelo. Mientras podamos permitírnoslo, deberemos utilizar el mayor número de individuos posible en la población, teniendo en cuenta, sin embargo, los límites físicos impuestos por la memoria de la GPU.

6.2 Tesis Erróneas

El algoritmo mejorado realiza migración entre el AG ejecutado sobre la CPU y el AG que utiliza la GPU, incluyéndose siempre el mejor individuo hasta el momento (élite) en el grupo de individuos intercambiados. Gracias a esta operación, los máximos valores de fitness obtenidos en los diferentes algoritmos (CPU y GPU) son muy parecidos. Sin embargo, observando los valores de aptitud obtenidos para las ejecuciones simples (sin paralelismo) de cada uno de los dos algoritmos, advertimos que el fitness que se obtiene en la versión que usa la GPU era muy bajo respecto a la versión CPU. Esto revela que el algoritmo GPU, por sí mismo, no es capaz de llegar a soluciones tan buenas como las que obtiene el algoritmo sin mejora. Sin embargo, cuando se realiza migración, es posible paliar deficiencias de este tipo gracias a que un algoritmo no muy bueno (en cuanto a la calidad de soluciones) recibe individuos muy buenos (producidos por un algoritmo que sí es capaz de obtener buenas soluciones) a partir de los cuales trabajar. Aunque el resultado final sea bueno, no puede alejarse demasiado de la calidad que se obtendría sin utilizar la mejora.

Si bien es cierto que nuestro objetivo inicial era reducir el tiempo de ejecución del AG tradicional sobre sistemas monoprocesadores, y no aumentar el fitness obtenido, sería conveniente alcanzar también este segundo objetivo. Si el AG sobre la GPU mejorase la calidad de sus soluciones, podría mejorar mucho las buenas soluciones logradas hasta el momento (tanto producidas en la GPU como recibidas mediante migración) y aportar a su vez mejores soluciones en la migración. Las soluciones finales del algoritmo global paralelo mejorarán si se cuenta con dos buenos algoritmos (en vez de con uno bueno y otro mediocre) en cuanto a la calidad de las soluciones.

La única diferencia esencial entre los dos algoritmos desarrollados, puesto que el resto de desigualdades se deben a que hay que salvar los detalles de estructura y funcionamiento característicos de la GPU, consiste en el mecanismo de obtención de números aleatorios.

El azar juega un papel fundamental en los AGs y, efectivamente, es de esperar que el uso de un juego de números *aleatorios* que en realidad no cumpla las propiedades estadísticas necesarias produzca malas soluciones.

6.2.1 El problema de la reordenación de los números aleatorios

Debido a la imposibilidad de obtener números aleatorios en la GPU, éstos deben ser generados en la CPU y llevados después a la memoria de la tarjeta (a fin de minimizar los accesos a la memoria externa a la GPU). La idea de la recombinación (reordenación) de números aleatorios, como medio para obtener nuevas poblaciones de números aleatorios con los que trabajar, precisa de una población inicial de aleatorios enorme.

Empíricamente se ha observado que, de no contar con poblaciones inmensas, las sucesivas reordenaciones van *uniformando* la población de aleatorios, de modo que tras pocas recombinaciones, la diversidad se ve reducida drásticamente, y no puede hablarse de una población de números aleatorios útil.

Una solución podría ser la de almacenar más números aleatorios en la textura (a modo de nuevas columnas asociadas a cada individuo), pero la memoria de la GPU es un recurso limitado que hay que robarle a la población de soluciones. De hecho, para la implementación correspondiente al problema defectivo solucionado (mapeo evolutivo), una población de 5000 individuos con una longitud de cromosoma de 100 ya no cabe en la tarjeta.

Planteamos aquí dos refinamientos del algoritmo GPU que se han implementado con éxito para todos los problemas (permiten mejorar el valor de fitness logrado):

Paralelización total del operador de mutación

Como vimos en el apartado 4.2.1 (estructuras de datos para la implementación sobre GPU), la mayor parte de los números aleatorios son consumidos por el operador genético de mutación. Con el fin de permitirse la mutación independientemente para cada gen de cada individuo, este operador utiliza un píxel de aleatorios por cada píxel de genes. Puesto que sólo contamos con un píxel de números aleatorios por individuo, este tipo de mutación nos obliga a recombinar tras mutar (en paralelo) cada columna de la zona de la textura correspondiente a los cromosomas.

Podemos obviar la recombinación de aleatorios para cada columna, y realizarla sólo tras la mutación de todas las columnas. Puesto que cada píxel de genes contiene cuatro genes en las cuatro componentes del mismo, y que cada componente se muta o no en función de la componente correspondiente del píxel de aleatorios, este refinamiento modificará el comportamiento del operador de mutación: Si se debe mutar un gen determinado, correspondiente a cierta componente del píxel, se mutarán todos los genes

que caigan en la misma componente para todos los píxeles del individuo. Es decir, la mutación de un gen dentro de un píxel implica la mutación de lcrom/4 genes en el cromosoma total del individuo. Para paliar este aumento del número de mutaciones, simplemente basta con disminuir la probabilidad de mutación (habría que estudiar para qué valor se obtienen los resultados óptimos).

Gracias a que no es necesario abandonar la GPU entre la mutación de dos columnas de la textura (para ejecutar el *kernel* encargado de la recombinación de aleatorios), el paralelismo de la operación de mutación es total (por filas y por columnas) en esta versión refinada del algoritmo GPU. El tiempo de ejecución mejora substancialmente, gracias a la eliminación de recombinaciones y al aumento del grado de paralelismo.

Refresco de los números aleatorios

La solución más eficiente y evidente pasa por generar en la CPU una nueva población de números aleatorios cuando la población residente en la memoria de la tarjeta ha perdido sus propiedades estadísticas.

El único problema de esta aproximación podría ser que la comunicación con la CPU fuera demasiado costosa y minase el excelente speed-up obtenido con el algoritmo sin refinar. Por otro lado, es de esperar una disminución de tiempo de ejecución debido a que se evitan lcrom-1 ejecuciones del *fragment program* encargado de la recombinación de números aleatorios, y a que la operación de mutación se realiza completamente en paralelo. De hecho, se han realizado algunas pruebas y se ha comprobado que el tiempo de ejecución disminuye para el algoritmo refinado, además de que la calidad de las soluciones obtenidas mejora significativamente (acercándose a la obtenida en el algoritmo CPU).

Parece que ésta es una buena línea de investigación a ampliar, localizando para qué frecuencia de refresco de aleatorios se obtiene los mejores resultados.

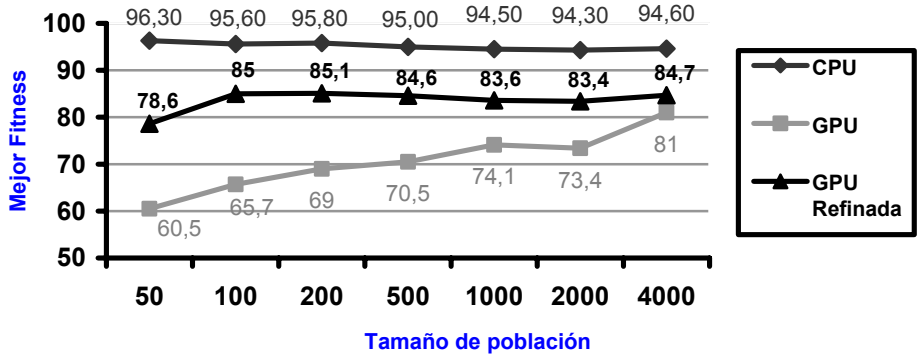
Primeros resultados

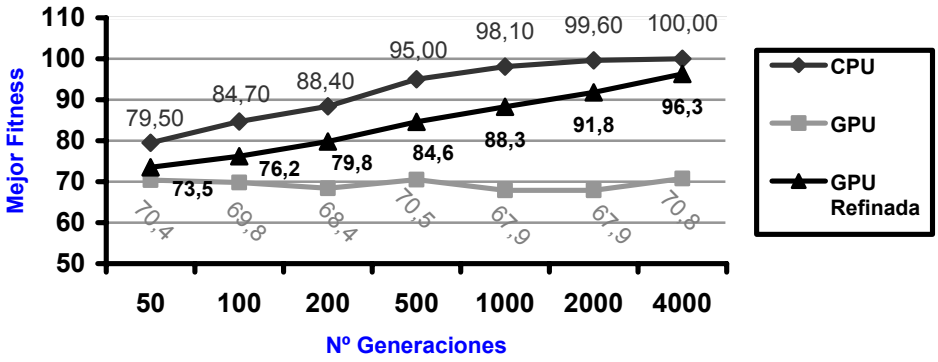
Se han implementado versiones del algoritmo refinado descrito para todos los problemas tratados (onemax, multimodal, defectivo y mapeo evolutivo), realizándose una batería de pruebas (de 10 ejecuciones para cada configuración), con un refresco de aleatorios cada 5 generaciones, para el problema onemax. Hemos obtenido los siguientes resultados de partida:

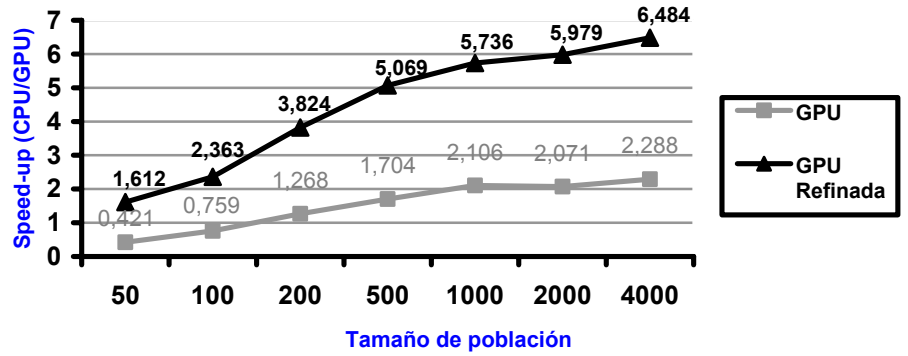
Tamaño Población	Fitness CPU	Fitness GPU	Fitness GPU refinado	Speed-up	Speed-up refinado
50	96,3	60,5	**78,6**	0,420756601840201	**1,61176048667612**
100	95,6	65,7	**85**	0,759436498693845	**2,36304552840714**
200	95,8	69	**85,1**	1,26779226165515	**3,82376535008746**
500	95	70,5	**84,6**	1,70407055629673	**5,06923066271704**
1000	94,5	74,1	**83,6**	2,10644591451084	**5,73564620725516**
2000	94,3	73,4	**83,4**	2,07127105096802	**5,97932495145359**
4000	94,6	81	**84,7**	2,28832242121958	**6,48373614945506**

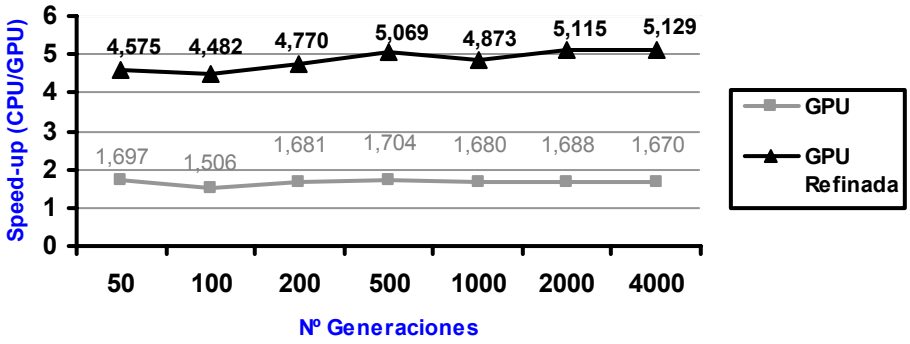
Número Generaciones	Fitness CPU	Fitness GPU	Fitness GPU refinado	Speed-up	Speed-up refinado
50	79,5	70,4	**73,5**	1,69738129099613	**4,57527186222172**
100	84,7	69,8	**76,2**	1,50556630881189	**4,48164151393225**
200	88,4	68,4	**79,8**	1,68135721096787	**4,76994211504592**
500	95	70,5	**84,6**	1,70407055629673	**5,06923066271704**
1000	98,1	67,9	**88,3**	1,67965075488178	**4,87334029505747**
2000	99,6	67,9	**91,8**	1,68805366808248	**5,11488879980915**
4000	100	70,8	**96,3**	1,6700536255323	**5,12890322140228**

Nótese que el speed-up referido aquí no es el correspondiente al algoritmo mejorado (AG paralelo), sino que se refiere a la relación entre los tiempos de ejecución del algoritmo simple CPU y del algoritmo simple GPU.









A la vista de los resultados obtenidos, el algoritmo refinado para los números aleatorios parece ser una excelente solución. No sólo aumenta notablemente los valores de fitness obtenidos, sino que el incremento del speed-up es sorprendente.

6.3 Trabajo Futuro

Ante los excelentes resultados obtenidos para los dos refinamientos indicados para mejorar la calidad de la población de números aleatorios residente en la tarjeta, se abre claramente una línea de futuras investigaciones en esta dirección: sería muy interesante estudiar qué frecuencia de regeneración consigue los mejores resultados, planteándose incluso la posibilidad de realizar un refresco tras cada iteración del bucle evolutivo, o incluso con mayor frecuencia (cada vez que se usen los cuatro valores de que dispone el píxel de aleatorios).

Sería conveniente profundizar en el estudio de la migración, y de los parámetros que puedan incidir en su coste (volumen de la población de intercambio, frecuencia, etc.), así como en la intervención del sistema operativo y su impacto, tanto en el coste de comunicación entre procesos, como en la planificación de la ejecución de los hilos paralelos.

La elección de los problemas abordados responde a la necesidad de comprobar, de una forma sencilla y directa, la bondad de los resultados obtenidos. Puesto que las funciones de fitness escogidas definen claramente la calidad de las soluciones logradas, y además resultan sencillas de implementar, se han tomado como una forma sencilla y segura de tantear las posibilidades de nuestra propuesta. Una vez comprobada las posibilidades que ofrece el modelo de paralelización planteado, el siguiente paso sería aplicarlo a problemas reales más complejos.

Aunque este estudio inicial ha demostrado un buen potencial para el modelo de paralelización propuesto, sería necesario ampliar los experimentos para entender y analizar el trabajo interno de los algoritmos.

Por último nos gustaría destacar que los resultados iniciales de este trabajo han sido presentados en el 1er Congreso de Informática (CEDI), celebrado en Granada del 13 al 16 de Septiembre de 2005, en la ponencia:

C. Córdoba, J.C. Pedraz, C. Tenllado, J.I. Hidalgo: "Hardware Gráfico Configurable: Una plataforma eficiente para la implementación de Algoritmos Genéticos". Actas del IV Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, págs. 829-836

APÉNDICE I Automatización de Pruebas

Para conseguir resultados fiables, necesitábamos ejecutar las distintas aplicaciones desarrolladas cambiando diferentes parámetros. Además, para cada configuración de parámetros, debíamos hacer varias ejecuciones, a fin conseguir datos más fiables.

Para ello se desarrolló una aplicación para cada problema genético, que automatiza la realización de ejecuciones, variando los parámetros de los algoritmos. El usuario introduce unos datos fijos para diversos parámetros de los AGs, y define varios tamaños y varias generaciones de CPU para realizar las pruebas.

Los resultados que se pretendían conseguir con estas pruebas era:

- Posiciones del mejores individuos.
- Mejores fitness.
- Tiempos de migración.
- Tiempos de ejecución.
- Número de generaciones ejecutadas por la GPU.
- Medias (aritméticas) de los resultados anteriores.
- Speed-ups.

El programa usa los parámetros fijos y variables de la entrada para crear los distintos algoritmos. Las pruebas que se realizan son de tres tipos:

1) **ejecución de algoritmos en paralelo**: se crea un algoritmo GPU y otro CPU, y se ejecutan en paralelo con migración de individuos entre ellos.
2) **ejecución de algoritmos simples**: se ejecutan los dos tipos de algoritmos por separado, para poder compararse entre ellos.
3) **ejecución de algoritmo de CPU comparativo**: se ejecuta un algoritmo de CPU con los parámetros de entrada y los resultados de las ejecuciones tipo 1. Esto nos permite comparar las ejecuciones en paralelo con la ejecución sobre la CPU con parámetros que nos aseguren un speed-up justo.

Como ya explicamos en el capítulo de resultados, el speed-up justo consiste en comparar el tiempo de ejecución de dos algoritmos en paralelo con un tamaño de población cualquier, con el tiempo de ejecución de un algoritmo de CPU con población la suma de las poblaciones de los algoritmos en paralelo, y como número de generaciones la suma de las generaciones realizadas por la ejecución paralela (generaciones de CPU más generaciones de GPU). Las generaciones de CPU vienen dadas por los datos de entrada del usuario, mientras que las de GPU las conseguimos de la ejecución en paralelo realizada con anterioridad.

Para cada una de estas pruebas se realiza un estudio por tamaños y otro por generaciones:

1) **Ejecuciones en paralelo**
   - Para cada uno de los tamaños de población, se lanza un número determinado de ejecuciones (repeticiones) de dos algoritmos en paralelo (cada uno con el tamaño dado), manteniendo constante el número de generaciones de la CPU (generaciones por defecto).
   - Para cada una de las generaciones, se lanza un número determinado de ejecuciones (repeticiones) de dos algoritmos en paralelo, manteniendo constante el tamaño de población de los algoritmos (el tamaño por defecto para cada uno de ellos).
2) **Ejecuciones simples**
   - Para cada uno de los tamaños de población, se lanza un número determinado de ejecuciones (repeticiones) de un algoritmo de CPU y otro de GPU por separado, manteniendo constante el número de generaciones.
   - Para cada uno de las generaciones, se lanza un número determinado de ejecuciones (repeticiones) de un algoritmo de CPU y otro de GPU por separado, manteniendo constante el tamaño de población de cada uno (por defecto).
3) **Ejecuciones comparativas**
   - Para cada uno de los tamaños de población multiplicado por dos, se lanza un número determinado de ejecuciones (repeticiones) de un algoritmo de CPU, con número de generaciones la suma de las generaciones de la ejecución correspondiente en paralelo. paralelo (generaciones GPU + generaciones CPU).
   - Para cada una de las generaciones, se lanza un número determinado de ejecuciones (repeticiones) de un algoritmo CPU, con tamaño de población por defecto multiplicado por dos y con generaciones la suma de las generaciones de la ejecución correspondiente en paralelo (generaciones GPU + generaciones CPU).

Los parámetros de los algoritmos genéticos son definidos mediante un fichero XML, que el usuario edita, mientras que los resultados de las ejecuciones se vuelcan a otro fichero XML, que estructura las pruebas y los datos resultantes.

A1.1 ¿Por qué XML?

A la hora de automatizar las pruebas, necesitábamos alguna manera de proporcionar datos para la aplicación, y que ella se ocupara de lanzar las distintas pruebas usando dichos datos. Además, queríamos hacer dos estudios, uno para generaciones y otro para tamaños de población, con lo que teníamos que replicar distintos datos para estos parámetros.

La mejor manera para hacer esto sin tener que tocar el código de cada aplicación era escribirlos en un fichero y que la aplicación los recogiese para ejecutar las pruebas. Además, los resultados debían ser volcados de igual manera para poder tener acceso a ellos siempre que quisiéramos, y no tener que copiarlos a mano de la salida por pantalla.

Sin embargo, la complicación surgía de la necesidad de estructurar dentro del fichero, tanto los datos de entrada como los resultados de salida. Sobre todo estos últimos, ya que serían muchos y complejos.

XML: un lenguaje de marcado

La mejor manera de estructurar un fichero de texto, según nuestra opinión, era usar un lenguaje de marcado que nos permitiera plasmar los datos indicando qué eran cada uno de ellos. Y XML nos proporciona estas facilidades.

XML es un lenguaje de marcado que nos permite crear nuestras propias etiquetas, que usamos para definir cada uno de los datos que queremos escribir. Además, estas etiquetas nos permiten construir una estructura anidada de todo el fichero, con lo que podemos desplegar aquellas etiquetas que nos interesen y ocultar las que no queramos [89].

La lectura y escritura de estos ficheros desde Visual C++ se hace mediante el uso de clases estándar que proporciona dicho lenguaje. Con ellas definimos lectores y escritores de estos ficheros, y los usamos para leer los datos según las etiquetas en las que se encuentran, y para escribir los datos englobados en nuestras propias etiquetas.

Así conseguimos una fácil lectura de los parámetros de entrada, así como una estructura de los resultados que nos facilita la búsqueda del dato que necesitamos [90].

A1.2 Clases Implementadas

Para el uso de XML se han implementado una serie de clases que nos dieran el soporte suficiente.

GestorXml

El GestorXml es el encargado de gestionar el código XML. Es quien recoge los datos de entrada leyéndolos del XML de entrada, y quien guarda los resultados de las ejecuciones. Además, es quien los vuelca al fichero de resultados.

Para mantener los resultados usa vectores. Estos vectores contienen objetos DatoSalida:

- Los resultados de las pruebas de migración por tamaños se guardan en el vector resultMigTams, por generaciones en el resultMigGens.
- Los resultados de las pruebas simples en resultSimpleTams y resultSimpleGens.
- Los de las comparativas de CPU en resultSpeedUpTams y resultSpeedUpGens.

Además, necesitamos los tiempos de ejecución de cada una de las pruebas. Estos se guardan en los vectores cuyo nombre comienza por time. El resto del nombre depende del tipo de estudio y del tipo de prueba. Por ejemplo, los tiempos de las pruebas de migración en paralelo para tamaños se guardan en timeMigTams. Estos vectores guardan floats.

Para ir tomando las medias de los resultados también necesitamos vectores. Estos empiezan por resultMid, seguidos de Mig, Simple o CPU según la prueba, y por Tams o Gens según el estudio. Estos vectores guardan objetos ResultMedio.

DatoSalida

Sirve para guardar los datos resultados de cada una de las repeticiones de las pruebas realizadas. Da soporte para el almacenamiento de la posición del mejor individuo de la población final de un algoritmo, su fitness, el tiempo de migración y el número de ejecuciones del AG.

Estos datos son necesarios para poder acceder a los datos necesarios para la ejecución de los CPU comparativos, y para el acceso a datos de información para estructurar correctamente el XML de salida.

ResultMedio

Sirve para guardar los datos de las medias de los resultados de las pruebas. Da soporte para el almacenamiento de tamaño de población, generaciones de CPU, de GPU y tiempo de ejecución, todos en media, de una prueba.

Estos resultados son las medias de los resultados de las repeticiones para una prueba realizada. Son necesarios para el cálculo a posteriori de speed-ups y acceso a datos de información para estructurar correctamente el XML de salida.

A1.3 XML de Entrada

Los datos de entrada son proporcionados por el usuario en el fichero "*entradaPruebas.xml*".

La estructura de este fichero es la siguiente:

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
    <Prob_Cruce>0,8</Prob_Cruce>
    <Prob_Mutacion>0,02</Prob_Mutacion>
    <Long_Cromosoma>100</Long_Cromosoma>
    <Migraciones>5</Migraciones>
    <Repeticiones>10</Repeticiones>
    <Tam_Defecto>500</Tam_Defecto>
    <Gens_Defecto>500</Gens_Defecto>
    <Tams_Poblacion>
        <tam>50</tam>
        <tam>100</tam>
        <tam>200</tam>
        <tam>500</tam>
```

```
        <tam>1000</tam>
        <tam>2000</tam>
        <tam>4000</tam>
    </Tams_Poblacion>
    <Generaciones>
        <gens>50</gens>
        <gens>100</gens>
        <gens>200</gens>
        <gens>1000</gens>
        <gens>2000</gens>
        <gens>4000</gens>
    </Generaciones>
</root>
```

Con esta estructura, permitimos al usuario fijar algunos parámetros del algoritmo para toda la batería de pruebas. Sin embargo, al mismo tiempo puede definir varios tamaños de población y de generaciones que debe realizar la CPU.

Los **parámetros fijos** para todas las ejecuciones, es decir, el valor indicado para este parámetro es fijo para todos los algoritmos lanzados en las pruebas, son:

- Prob_cruce: establece la probabilidad de cruce de los AG.
- Prob_Mutación: establece la probabilidad de mutación de los AG.
- Long_Cromosoma: establece longitud del cromosoma de los individuos de los AG.
- Migraciones: establece el número de migraciones en la ejecución paralela de los AG.
- Repeticiones: establece el número de ejecuciones a realizar para una misma configuración de parámetros.
- Tam_Defecto: establece el tamaño por defecto de la población del AG. Ésta será la población de los algoritmos en las pruebas simples y en paralelo, en el estudio por generaciones.
- Gens_Defecto: establece las generaciones por defecto de los algoritmos simples, y el algoritmo CPU en las pruebas paralelas, en el estudio por tamaños.

Los **parámetros variables** son:

- Tams_Poblacion: agrupa los distintos tamaños de población para cada uno de los algoritmos de las pruebas en paralelo y simples. Cada uno de ellos se define en la etiqueta "tam".
- Generaciones: agrupa los distintos números de generaciones para cada uno de los algoritmos simples y para el algoritmo de CPU en las pruebas en paralelo.

> ! *Es importante tener en cuenta que el XML tiene **codificación UTF-8**, con lo que no se podrán usar ciertos caracteres como "ñ" o tildes. Además, dependiendo de la configuración de moneda y decimales del sistema operativo en que se ejecute, el separador decimal deberá ser "," ó ".".*

A1.4 Funcionamiento

La batería de pruebas se realiza en el método *main* de la aplicación. Básicamente, este main creará y lanzará varios algoritmos genéticos (CPU o GPU), dependiendo del tipo de prueba.

Lo primero que se hacen son las pruebas con algoritmos en paralelo. Para ello se crean dos algoritmos, uno de CPU y otro de GPU; y se ejecutan en paralelo con migración. Los resultados de fitness, generaciones ejecutadas por GPU, tiempos de ejecución y migración, etc…, se guardan para más adelante volcarlos al XML de salida, y para ser usados en las pruebas comparativas.

Después se hacen las pruebas con algoritmos simples. Para ello creamos un algoritmo de CPU y otro de GPU, y los ejecutamos por separado. Primero se lanza uno y se recogen los datos, después el otro y también se recogen.

Por último realizamos las pruebas comparativas de algoritmos de CPU. Estas pruebas lanzan un solo algoritmo genético, de tipo CPU, con los valores idóneos que nos aseguren un speed-up justo al comparar el tiempo de ejecución con el de las pruebas en paralelo. Para esto necesitamos los resultados obtenidos para estos últimos.

A1.4.1 Pruebas en Paralelo

Realizamos dos tipos de estudios lanzando dos AGs y ejecutándolos en paralelo con migración entre ellos: por tamaños y por generaciones.

Estudio por tamaños

Para cada tamaño de población de entrada, creamos dos algoritmos, uno CPU y otro GPU, cuya población será indicada por la entrada, y asignamos al CPU las generaciones por defecto del XML de entrada. Para el resto de parámetros de los parámetros usamos los datos fijos del XML.

A continuación lanzamos la ejecución de ambos en paralelo y recogemos los datos de salida. Una vez hecho esto los destruimos.

Este proceso se repite tantas veces como repeticiones indiquemos en el XML de entrada, para los mismos valores de los parámetros del XML.

Por ejemplo, para el XML anterior, lanzaríamos 10 ejecuciones de dos AGs (uno CPU y otro GPU) en paralelo, cuya población sería 50 individuos para cada uno y las generaciones del CPU serían 500. Después haríamos lo mismo pero con el tamaño de población 100 para cada algoritmo. Así seguiríamos para todos los tamaños especificados.

Estudio por generaciones

Para cada generación de entrada, creamos dos algoritmos, uno CPU y otro GPU, cuya población será la de por defecto de la entrada para cada uno; y asignamos al CPU las

generaciones de la entrada. Para el resto de parámetros de los parámetros usamos los datos fijos del XML.

A partir de aquí seguimos el mismo proceso que en el estudio anterior, ejecutando para cada generación distinta el número de repeticiones de la entrada, manteniendo constantes el resto de parámetros.

Por ejemplo, para el XML anterior, lanzaríamos 10 ejecuciones de dos AGs (uno CPU y otro GPU) en paralelo, cuya población sería 500 individuos para cada uno y las generaciones del CPU serían 50. Después haríamos lo mismo pero con las generaciones de CPU 100. Así seguiríamos para todas las generaciones de la entrada.

A1.4.2 Pruebas Simples

En esta ocasión lanzamos los algoritmos para su ejecución en solitario. Es decir, que no hay paralelismo entre ellos, y por tanto no hay migración. Esto nos permite compararlos entre ellos.

El estudio que realizamos también es por tamaños y generaciones

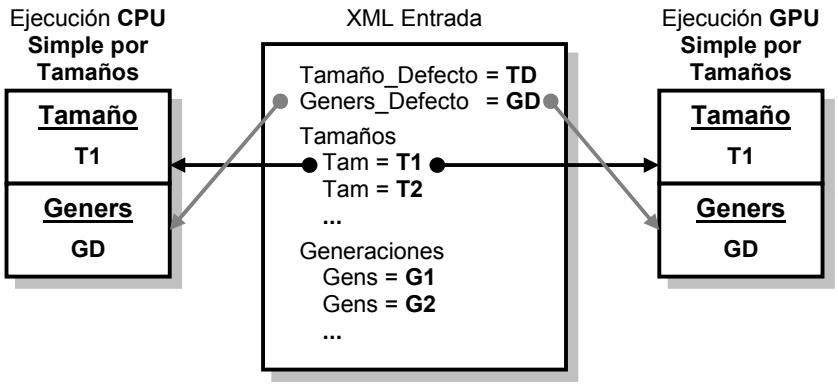
Estudio por tamaños

Para cada tamaño de población de entrada, creamos dos algoritmos, uno CPU y otro GPU, cuya población será indicada por la entrada, y asignamos a ambos las generaciones por defecto del XML de entrada. Para el resto de parámetros de los parámetros usamos los datos fijos del XML.

A continuación lanzamos la ejecución del algoritmo de CPU y recogemos los datos de salida. Una vez hecho esto lo destruimos. Después lanzamos la ejecución del GPU y tomamos los resultados. Una vez acabado lo destruimos.

Este proceso se repite tantas veces como repeticiones indiquemos en el XML de entrada, para los mismos valores de los parámetros del XML.

Por ejemplo, para el XML anterior, lanzaríamos 10 ejecuciones de dos AGs (uno CPU y otro GPU) por separado, cuya población sería 50 individuos para cada uno y las generaciones de ambos serían 500. Después haríamos lo mismo pero con el tamaño de población 100 para cada algoritmo. Así seguiríamos para todos los tamaños especificados.

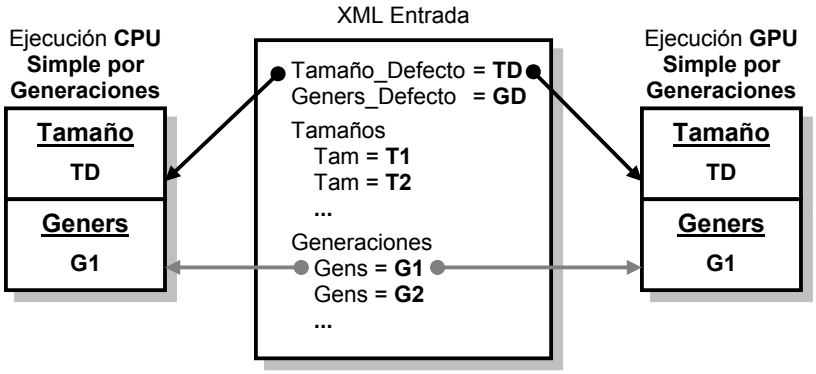


Estudio por generaciones

Para cada generación de entrada, creamos dos algoritmos, uno CPU y otro GPU, cuya población será la de por defecto de la entrada para cada uno; y asignamos a ambos las generaciones de la entrada. Para el resto de parámetros de los parámetros usamos los datos fijos del XML.

A partir de aquí seguimos el mismo proceso que en el estudio anterior, ejecutando para cada generación distinta el número de repeticiones de la entrada, manteniendo constantes el resto de parámetros.

Por ejemplo, para el XML anterior, lanzaríamos 10 ejecuciones de dos AGs (uno CPU y otro GPU) por separado, cuya población sería 500 individuos para cada uno y las generaciones de ambos serían 50. Después haríamos lo mismo pero con generaciones igual a 100. Así seguiríamos para todas las generaciones de la entrada.



A1.4.3 Pruebas de CPU Comparativas

En esta ocasión lanzamos un solo algoritmo CPU para su ejecución. El estudio que realizamos también es por tamaños y generaciones.

Estas pruebas se realizan para comparar la mejora de la introducción del algoritmo de GPU en la ejecución de pruebas en paralelo. Queremos comprobar que es mejor la ejecución de dos algoritmos en paralelo (uno de ellos GPU) con la población repartida, que un único algoritmo de CPU con la población completa. Por tanto, el tamaño de las poblaciones de los algoritmos ejecutados en estas pruebas serán el doble que el de sus correspondientes en paralelo.

Además, el número de generaciones que deben ejecutar es la suma de las generaciones ejecutadas en sus correspondientes algoritmos en paralelo.

Esto quedará más claro explicando los estudios que realizamos y viendo un ejemplo con el XML indicado en el apartado 4.1.

Estudio por tamaños

Para cada tamaño de población de entrada, creamos un algoritmo de CPU, cuya población será el doble de la indicada por la entrada; y le asignamos la suma de

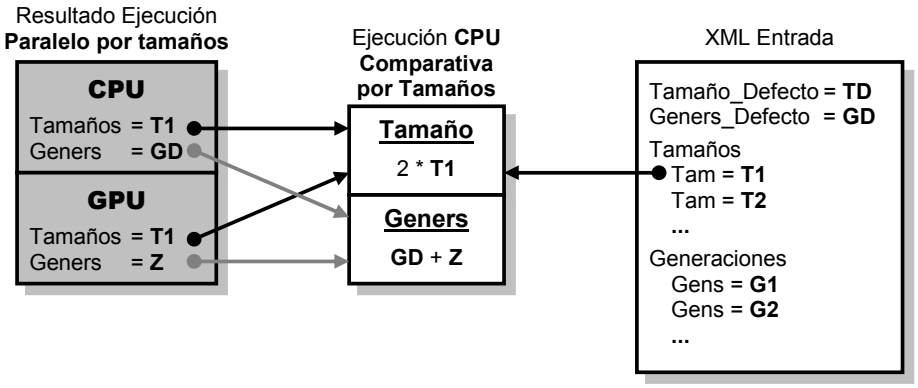
generaciones ejecutadas en sus correspondientes algoritmos en paralelo. Para el resto de parámetros de los parámetros usamos los datos fijos del XML.

A continuación lanzamos la ejecución del algoritmo de CPU y recogemos los datos de salida. Una vez hecho esto lo destruimos.

Este proceso se repite tantas veces como repeticiones indiquemos en el XML de entrada. En contraposición a las pruebas anteriores, las generaciones de las siguientes repeticiones no son las mismas que en la primera ejecución. Esto se debe a que tenemos un número de repeticiones de ejecuciones de algoritmos en paralelo para los mismos parámetros de entrada. Cada una de ellas tendrá un número de generaciones ejecutadas distinto. Queremos usar estas generaciones distintas en las repeticiones de los CPU comparativas.

Por ejemplo, para el XML anterior, lanzaríamos una ejecución de un AG de CPU, cuya población sería 100 individuos y las generaciones serían la suma de generaciones de la repetición 0 de la ejecución en paralelo de algoritmos con población 50 para cada uno y generaciones de CPU 500. Después haríamos lo mismo pero con la repetición 1 de los paralelos. Continuamos así hasta 10 repeticiones.

A continuación hacemos lo mismo para el tamaño de población 200 y así seguiríamos para todos los tamaños especificados.



Estudio por generaciones

Para cada generación de entrada, creamos un algoritmo de CPU, cuya población será el doble de la de por defecto de la entrada; y le asignamos la suma de generaciones ejecutadas en sus correspondientes algoritmos en paralelo. Para el resto de parámetros de los parámetros usamos los datos fijos del XML.

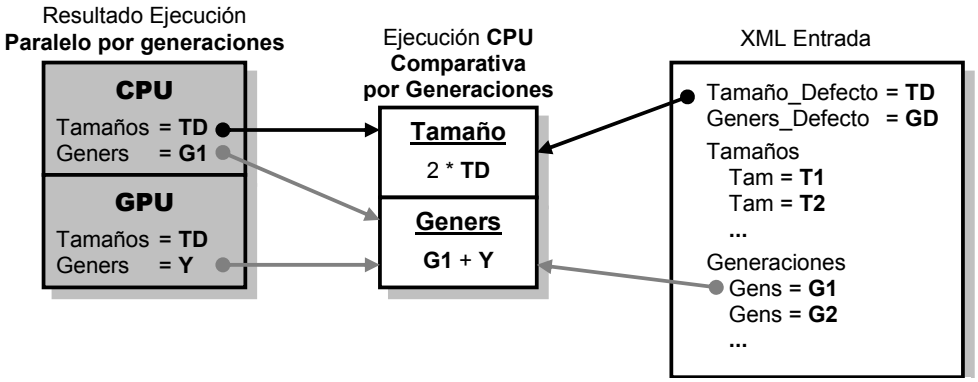
A continuación lanzamos la ejecución del algoritmo de CPU y recogemos los datos de salida. Una vez hecho esto lo destruimos.

Este proceso se repite tantas veces como repeticiones indiquemos en el XML de entrada. En este caso también hacemos corresponder las generaciones de cada repetición con las de la repetición correspondiente en paralelo.

Por ejemplo, para el XML anterior, lanzaríamos una ejecución de un AG de CPU, cuya población sería 1000 individuos y las generaciones serían la suma de generaciones de la repetición 0 de la ejecución en paralelo de algoritmos con generaciones de CPU 50.

Después haríamos lo mismo pero con la repetición 1 de los paralelos. Continuamos así hasta 10 repeticiones.

A continuación hacemos lo mismo para las generaciones de CPU 100 y así seguiríamos para todos los tamaños especificados.



A1.5 XML de Salida

Para poder acceder cómodamente a los resultados de todas las pruebas anteriores, volcamos los datos en un XML. A pesar de todo, la cantidad de pruebas realizadas suponen una gran cantidad de información que debe ser estructurada convenientemente.

A continuación mostraremos paso a paso el aspecto de este fichero, con nombre "log_*problema*.xml", donde *problema* es el nombre del problema genético a resolver.

El aspecto inicial, sin desplegar, del XML es el siguiente:

```
<?xml version="1.0" encoding="utf-8" ?>
-<root>
    <Prob_Cruce>0,8</Prob_Cruce>
    <Prob_Mutacion>0,02</Prob_Mutacion>
    <Long_Cromosoma>100</Long_Cromosoma>
    +<Ejecuciones>
    +<SpeedUps>
</root>
```

Podemos ver que tras la cabecera del XML aparece la etiqueta *root*, que engloba todos los datos del fichero. A continuación aparecen los datos de los AGs que permanecen fijos para todos los resultados de la batería.

Bajo la etiqueta *Ejecuciones* encontraremos todas las ejecuciones de pruebas realizadas, y bajo la etiqueta *SpeedUps* los resultados comparativos.

Conviene decir que, a medida que vayamos entrando en nodos anidados, los parámetros de los AGs que nos dicen los valores con que éstos han sido ejecutados van apareciendo, y son válidos para toda la zona anidada.

A1.5.1 Ejecuciones

Bajo este epígrafe están todos los resultados de las ejecuciones realizadas:

```
-<Ejecuciones>
   +<Migrando>
   +<Simple>
   +<CPU_Comparativas>
</root>
```

Aquí vemos los tres tipos de pruebas realizadas. Básicamente, la estructura interna de las tres etiquetas es muy similar.

Migrando

Bajo esta etiqueta se agrupan los resultados de las pruebas de AGs ejecutados en paralelo con migración entre ellos.

```
-<Migrando>
   <Migraciones>5</Migraciones>
   +<Por_Tams>
   +<Por_Gens>
</Migrando>
```

Aquí podemos ver el número de migraciones que realizan las ejecuciones en paralelo. Debajo de cada una de las etiquetas desplegables *Por_Tams* y *Por_Gens* aparecen los resultados agrupados según el estudio sea por tamaños o por generaciones.

Por_Tams

Aparecen los resultados del estudio por tamaños.

```
-<Por_Tams>
   <Gens_CPU>500</Gens_CPU>
   +<Bateria>
   +<Bateria>
</Por_Tams>
```

Vemos las generaciones del algoritmo de CPU ejecutadas para cada prueba. Cada batería contiene los resultados de las repeticiones para un tamaño de población de entrada distinto.

Batería

Aparecen los resultados para un tamaño de población de la entrada.

```
-<Bateria>
   <Tam_Poblacion>50</Tam_Poblacion>
   +<Resultado>
   +<Resultado>
   +<Medias>
</Bateria>
```

Nos informan del tamaño de la población usado para cada uno de los AGs paralelos. Bajo *Resultados* encontraremos los datos de salida para cada repetición. Es decir, que habrá tantas etiquetas *Resultado* como repeticiones hayamos indicado en el XML de entrada.

Medias contiene las medias de los resultados de la batería.

Resultado

Aparecen los resultados para una repetición de la ejecución.

```
-<Resultado>
<CPU>
    <Mejor>0</Mejor>
    <Fitness>90</Fitness>
    <Tiempo_Migracion>1169,41553853452</Tiempo_Migracion>
</CPU>
<GPU>
    <Mejor>0</Mejor>
    <Fitness>54</Fitness>
    <Tiempo_Migracion>505,480051881363</Tiempo_Migracion>
    <Gens_GPU>312</Gens_GPU>
</GPU>
<Tiempo_Ejecucion>1221972,95202964</Tiempo_Ejecucion>
</Resultados>
```

Vemos los resultados de ambos algoritmos, en el que nos indican la posición del mejor individuo en la población, su fitness (el mejor de todos los individuos) y el tiempo de migración total consumido en la ejecución de la prueba.

Para la GPU conseguimos, además, el número de generaciones ejecutadas.

Para terminar, aparece el tiempo de ejecución de los dos algoritmos en paralelo.

Medias

Aparece la media de los resultados obtenidos en las repeticiones.

```
-<Medias>
    <CPU>
        <Mejor_Fitness>0</Mejor_Fitness>
        <Fitness _Medio>90,6</Fitness_Medio>
        <T_Migración_Medio>2743,41553853452</T_Migración_Medio>
    </CPU>
    <GPU>
        < Mejor_Fitness >70</ Mejor_Fitness >
        < Fitness_Medio >62,9</ Fitness_Medio >
        < T_Migración_Medio >517,480051881363</ T_Migración_Medio >
        <Gens_GPU_Medio >165</Gens_GPU_Medio >
    </GPU>
    <T_Ejecucion_Medio >1232248,95202964</T_Ejecucion_Medio >
</Medias>
```

Vemos el fitness mejor, el medio, y el tiempo de migración medio para cada AG. También las generaciones medias que ejecuta la GPU, y el tiempo de ejecución medio de las pruebas.

Por_Gens

Aparecen los resultados del estudio por generaciones.

```
-<Por_Gens>
    <Tam_Poblacion>1000</Tam_Poblacion>
    +<Bateria>
    +<Bateria>
</Por_Gens>
```

Vemos el tamaño de población para cada uno de los algoritmos en paralelo de las pruebas realizadas. Cada batería contiene los resultados de las repeticiones para una generación de CPU de entrada distinto.

Batería

Aparecen los resultados para una generación de CPU de la entrada.

```
-<Bateria>
    <Gens_CPU>50</Gens_CPU>
    <Frec_Migracion>9</Frec_Migracion>
    +<Resultado>
    +<Resultado>
    +<Medias>

</Bateria>
```

Nos informan de las generaciones ejecutadas para el algoritmo de CPU. Bajo *Resultados* encontraremos los datos de salida para cada repetición. Es decir, que habrá tantas etiquetas *Resultado* como repeticiones hayamos indicado en el XML de entrada.

Medias contiene las medias de los resultados de la batería.

Resultado

Aparecen los resultados para una repetición de la ejecución.

La estructura es la misma que para tamaños.

Medias

Aparecen las medias de los resultados para las pruebas.

La estructura es la misma que para los tamaños.

Simple

Bajo esta etiqueta se agrupan los resultados de las pruebas de AGs ejecutados por separado.

```
-<Simple>
    +<Por_Tams>
    +<Por_Gens>
</Simple>
```

Debajo de cada una de las etiquetas desplegables *Por_Tams* y *Por_Gens* aparecen los resultados agrupados según el estudio sea por tamaños o por generaciones.

El resto de etiquetas tienen la misma estructura que en el caso de *Migrando*. Las únicas diferencias residen en que en esta ocasión, las generaciones que se indican se ejecutan tanto en CPU como en GPU, ya que los algoritmos son lanzados por separado, y ambos terminan cuando llegan a las generaciones especificadas.

Por tanto, no existen *Gens_GPU* ni *Gens_GPU_Medio*.

CPU_Comparativas

Bajo esta etiqueta se agrupan los resultados de las pruebas de AG de CPU con datos comparativos.

```
-<CPU_Comparativas>
    +<Por_Tams>
```

```
    +<Por_Gens>
</CPU_Comparativas>
```

Debajo de cada una de las etiquetas desplegables *Por_Tams* y *Por_Gens* aparecen los resultados agrupados según el estudio sea por tamaños o por generaciones.

Por_Tams

Aparecen los resultados del estudio por tamaños.

```
-<Por_Tams>
    +<Bateria>
    +<Bateria>
</Por_Tams>
```

Cada batería contiene los resultados de las repeticiones para un tamaño de población de entrada distinto.

Batería

Aparecen los resultados para un tamaño de población de la entrada.

```
-<Bateria>
    <Tam_Poblacion>100</Tam_Poblacion>
    +<Resultado>
    +<Resultado>
    +<Medias>
</Bateria>
```

Nos informan del tamaño de la población usado para el AG. Se puede ver que es el doble del primer tamaño de entrada.

Bajo *Resultados* encontraremos los datos de salida para cada repetición. Es decir, que habrá tantas etiquetas *Resultado* como repeticiones hayamos indicado en el XML de entrada.

Medias contiene las medias de los resultados de la batería.

Resultado

Aparecen los resultados para una repetición de la ejecución.

```
-<Resultado>
<Gens>821</Gens>
<Mejor>0</Mejor>
<Fitness>89</Fitness>
<Tiempo_Ejecucion>1525323,95202964</Tiempo_Ejecucion>
</Resultados>
```

Aparecen las generaciones ejecutadas por el algoritmo (suma de ejecuciones de la repetición correspondiente a la ejecución paralela de algoritmos en igualdad de condiciones), el mejor individuo, su fitness y el tiempo de ejecución del algoritmo.

Medias

Aparece la media de los resultados obtenidos en las repeticiones.

```
-<Medias>
    <Gens_Medio >165</Gens_Medio >
    <Mejor_Fitness>0</Mejor_Fitness>
```

```
    <Fitness_Medio>90,6</Fitness_Medio>
    <T_Migracion_Medio>2743,41553853452</T_Migracion_Medio>
    <T_Ejecucion_Medio >1232248,95202964</T_Ejecucion_Medio >
</Medias>
```

Vemos las generaciones medias ejecutadas, el fitness mejor, el medio, y el tiempo de migración medio del AG.

Por_Gens

Aparecen los resultados del estudio por generaciones.

```
-<Por_Gens>
    <Tam_Poblacion>2000</Tam_Poblacion>
    +<Bateria>
    +<Bateria>
</Por_Gens>
```

Vemos el tamaño de población para del algoritmo (el doble de por defecto). Cada batería contiene los resultados de las repeticiones para las generaciones ejecutadas en la repetición paralela correspondiente en igualdad de condiciones.

Batería

Aparecen los resultados para una generación de CPU de la entrada.

```
-<Bateria>
    +<Resultados>
    +<Resultados>
    +<Medias>
</Bateria>
```

Bajo *Resultados* encontraremos los datos de salida para cada repetición. Es decir, que habrá tantas etiquetas *Resultados* como repeticiones hayamos indicado en el XML de entrada.

Resultados

Aparecen los resultados para una repetición de la ejecución. La estructura es la misma que para tamaños.

Medias

Aparecen las medias de los resultados para las pruebas.

La estructura es la misma que para los tamaños.

A1.5.2 Speed-ups

Contiene los speed-up resultantes de las ejecuciones anteriores.

```
-<SpeedUps>
    +<Simples>
    +<Hilos>
</SpeedUps>
```

Los simples son los speed-ups de las comparaciones de los algoritmos CPU y GPU ejecutados por separado. Con ellos podemos medir la mejora del rendimiento de un AG ejecutado en la tarjeta respecto al mismo AG ejecutado en CPU.

Los hilos son los speed-ups de las comparaciones de los algoritmos CPU y GPU ejecutados en paralelo con el algoritmo CPU en igualdad de condiciones. Con ellos podemos medir la mejora del rendimiento de dos AGs en paralelo con población distribuida frente a un AG con la población completa.

Simples

Aparecen los speed-ups agrupados en tamaños y generaciones.

```
-<Simples>
   -<Por_Tams>
      <Gens>500</Gens>
      +<Resultado>
      +<Resultado>
   </Por_Tams>
   -<Por_Gens>
      <Tam_Poblacion>500</Tam_Poblacion>
      +<Resultado>
      +<Resultado>
   </Por_Gens>
</Simples>
```

Por tamaños vemos las generaciones ejecutadas para los algoritmos de todas las pruebas, mientras que por generaciones vemos el tamaño de población de los algoritmos para todas las pruebas.

Resultado tendrá los resultados de speed-up para cada tamaño de población (en el estudio por tamaños) y para cada generación (en el estudio de generaciones).

Resultado

Para tamaños:

```
-<Resultado>
   <Tam_Pob>50</Tam_Pob>
   <CPU_time>446723,2355</CPU_time>
   <GPU_time>1130995,649093</GPU_time>
   <SpeedUp>0,412665</SpeedUp>
</Resultado>
```

Aparece el tamaño de población de los algoritmos comparados. Los resultados son el tiempo de ejecución del AG de CPU, el del AG de GPU, y el speed-up resultante.

En el caso de generaciones, la única diferencia es que *Tam_Pob* se cambia por *Gens*, que nos indica el número de generaciones ejecutadas en cada algoritmo.

Hilos

Aparecen los speed-ups agrupados en tamaños y generaciones.

```
-<Hilos>
   -<Por_Tams>
      +<Resultado>
      +<Resultado>
   </Por_Tams>
   -<Por_Gens>
```

```
        +<Resultado>
        +<Resultado>
    </Por_Gens>
</Hilos>
```

En ambos casos sólo aparecen los resultados obtenidos.

Como ya hemos explicado, por tamaños comparamos la ejecución de dos AGs paralelos con un tamaño de población de entrada para cada uno de ellos, y como generaciones del CPU las generaciones por defecto; frente a un algoritmo de CPU con tamaño de población la suma de los anteriores (el doble para cada uno de los tamaños de entrada) y con generaciones la suma de las generaciones ejecutadas en los paralelos.

En el caso de generaciones, comparamos la ejecución de dos AGs paralelos con tamaño de población por defecto en cada uno de ellos, y con una generación de entrada para el CPU; frente a un algoritmo de CPU con población la suma de los tamaños de los paralelos (el doble de la de por defecto), y con generaciones la suma de generaciones de los anteriores (generaciones por defecto más generaciones ejecutadas en GPU).

Resultado

Por_Tams

Aparecen los datos de las ejecuciones que se comparan y sus resultados comparados.

```
-<Resultado>
    <Tam_Pob_Mig>50</Tam_Pob_Mig>
    <Tam_Pob_Hilo_CPU>100</Tam_Pob_Hilo_CPU>
    <Gens_CPU_Mig>500</Gens_CPU_Mig>
    <Gens_GPU_Mig>165</Gens_GPU_Mig>
    <Gens_Hilo_CPU>665,2</Gens_Hilo_CPU>
    <Mig_time>1232248,0340148</Mig_time>
    <CPU_time>1244134,54592197</CPU_time>
    <SpeedUp>1,00964620074779</SpeedUp>
</Resultado>
```

Vemos por orden: el tamaño de población de cada AG ejecutado en paralelo (*Tam_Pob_Mig*), el tamaño del AG de CPU comparativo (*Tam_Pob_Hilo_CPU*), las generaciones de CPU y GPU para los paralelos (*Gens_CPU_Mig* y *Gens_GPU_Mig*), las generaciones ejecutadas en el CPU comparativo (*Gens_Hilo_CPU*), el tiempo de ejecución de la prueba en paralelo (*Mig_time*), y el tiempo de ejecución del CPU comparativo (*CPU_time*).

Por último, aparece el speed-up conseguido.

Por_Gens

Aparecen los datos de las ejecuciones que se comparan y sus resultados comparados.

```
-<Resultado>
    <Gens_CPU_Mig>500</Gens_CPU_Mig>
    <Gens_GPU_Mig>165</Gens_GPU_Mig>
    <Gens_Hilo_CPU>665,2</Gens_Hilo_CPU>
    <Tam_Pob_CPU_Mig>50</Tam_Pob_CPU_Mig>
    <Tam_Pob_GPU_Mig>50</Tam_Pob_GPU_Mig>
    <Tam_Pob_Hilo_CPU>100</Tam_Pob_Hilo_CPU>
    <Mig_time>1232248,0340148</Mig_time>
    <CPU_time>1244134,54592197</CPU_time>
    <SpeedUp>1,00964620074779</SpeedUp>
</Resultado>
```

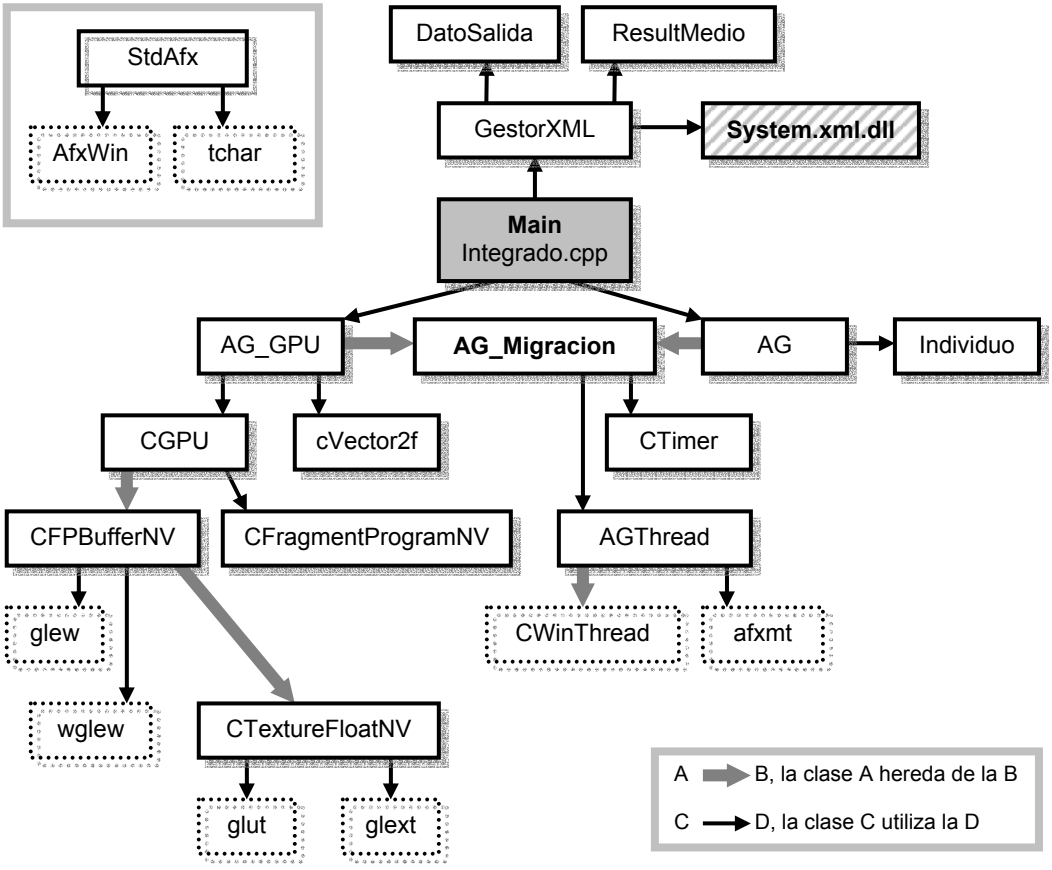
Vemos por orden: las generaciones de CPU y GPU para los paralelos (*Gens_CPU_Mig* y *Gens_GPU_Mig*), las generaciones ejecutadas en el CPU comparativo (*Gens_Hilo_CPU*), el tamaño de población del AG de CPU ejecutado en paralelo (*Tam_Pob_CPU_Mig*), el tamaño de población del AG de GPU ejecutado en paralelo (*Tam_Pob_GPU_Mig*), el tamaño del AG de CPU comparativo (*Tam_Pob_Hilo_CPU*), el tiempo de ejecución de la prueba en paralelo (*Mig_time*), y el tiempo de ejecución del CPU comparativo (*CPU_time*).

Por último, aparece el speed-up conseguido.

APÉNDICE II Jerarquía de Clases

Incluimos este apéndice a modo de documentación adicional de implementación, orientada principalmente a posibles usuarios de las aplicaciones desarrolladas. No entraremos en excesivo detalle, sino que nos limitaremos a describir la jerarquía de clases implementada y las características y utilidades ofrecidas por cada una de estas clases. Para aquellos que deseen modificar o estudiar más a fondo la implementación, remitimos a la intensiva documentación del propio código (incluido en el CD adjunto).

La jerarquía de clases desarrollada se representa en el siguiente esquema, donde se señala qué clases usan a cuáles (➔) y qué clases heredan de cuáles (➞):



Las clases encuadradas en línea discontinua son clases estándar ya existentes de C++ y de las librerías OpenGL.

La clase StdAfx es incluida por todas las demás, ya que se trata del archivo de cabecera de precompilado. Este fichero contiene todas las directivas de inclusión básicas que necesitan hacer la mayoría de las clases del proyecto. Puesto que es el primer fichero que se compila, podemos asegurarnos de que ciertos archivos de cabecera se compilan antes que los demás, simplemente añadiendo sus inclusiones en la clase StdAfx. De este modo se pueden solucionar determinados problemas de linkado (ver apartado A3.1 del Apéndice III).

La clase Integrado es la que contiene el método main() de la aplicación. En esencia, crea los objetos correspondientes a los AGs a usar (con los valores de parámetros adecuados) y los lanza en sus respectivos procesos. El tipo de AGs a usar, así como la fuente de la que se obtienen los valores de los parámetros (usuario o fichero XML para batería de pruebas) dependen de la aplicación concreta.

En cualquiera de las implementaciones, la clase principal Integrado hace uso de AG y AG_GPU, las dos clases más importantes de toda la jerarquía.

Ambas clases derivan de la superclase AG_Migracion, que contiene los campos y métodos necesarios para realizar migración con otro AG (también instancia de AG_Migracion). Esta clase es en realidad una carcasa hueca que sólo desarrolla esta funcionalidad común. Utiliza la clase CTimer para medir el tiempo de comunicación entre las dos poblaciones que intercambian individuos, y AGThread como soporte de hilos (los AGs que migran se ejecutan en procesos diferentes).

AGThread hereda de la clase estándar de Windows para hilos (CWinThread).

CTimer es la clase de medición de tiempos implementada para tomar tiempos de ejecución de los diferentes AGs, así como de algunas partes concretas del algoritmo (como por ejemplo la comunicación entre AGs).

Clases AG e Individuo

AG es la clase correspondiente a la implementación del algoritmo genético que se ejecuta exclusivamente sobre la CPU (algoritmo no mejorado) y, por tanto, se corresponde con la implementación tradicional en C++ de un AG simple. Se corresponde con la descripción incluida en el apartado 4.1 de implementación sobre la CPU. El método principal realiza el bucle de evolución (aplicando los correspondientes operadores genéticos) sobre la población. Recordamos que la población consiste en un vector de punteros a individuos, cada uno de los cuales es una instancia de la clase Individuo.

La clase Individuo implementa el tipo de datos del individuo como un vector de enteros (cromosoma binario), y contiene métodos para manejar dicha estructura y algunos datos auxiliares necesarios en el proceso evolutivo implementado en la clase AG.

Clase AG_GPU

Es la correspondiente a la implementación del algoritmo genético que se ejecuta exclusivamente sobre la GPU (parte mejorada del algoritmo, que no el algoritmo mejorado). Se corresponde con la descripción incluida en el apartado 4.2 de

implementación sobre la GPU. El método principal realiza el bucle de evolución en el que se van ejecutando los *fragment programs* correspondientes a los operadores genéticos sobre la población, que en este caso está en la tarjeta gráfica (no en la memoria principal del sistema).

AG_GPU utiliza la clase auxiliar cVector2f (vectores formados por dos valores float) y la clase CGPU.

CGPU es la clase encargada de interaccionar con el buffer de la GPU (inicialización, lectura y escritura, etc.). Utiliza CFragmentProgramNV para manejar los *fragment programs* (carga, vinculación, paso de parámetros, etc.), y hereda de la clase CFPBufferNV, que trabaja a un nivel más bajo, usando las extensiones de OpenGL glew y wglew.

CTextureFloatNV es la superclase de la que hereda CFPBufferNV (y de ésta, a su vez, hereda CGPU), y utiliza las extensiones OpenGL glut y glext. Ésta es la clase que inicializa la textura y la vincula, aunque AG_GPU trabaje con la clase CGPU por ser más intuitiva y fácil de manejar.

Clases GestorXML y auxiliares

Las clases GestorXML, DatoSalida y ResultMedio corresponden a la parte de automatización de pruebas, que se realiza mediante ficheros XML (tanto para parametrización de los algoritmos genéticos como para obtención de resultados). La explicación detallada de estas clases puede encontrarse en el Apéndice I.

APÉNDICE III Detalles de Implementación

A la hora de desarrollar las aplicaciones realizadas, nos encontramos con distintos problemas derivados del propio lenguaje de implementación, tanto en Visual C++ .NET como con Cg.

En este apartado vamos a ir detallando distintas complicaciones que surgieron en la fase de implementación, y explicar cómo conseguimos resolverlas. Esperamos que este apéndice sea de utilidad en la resolución de problemas para otras personas que desarrollen aplicaciones parecidas.

A3.1 Problemas de Generación y Linkado

Cuando desarrollamos una aplicación en Visual C++ .NET, tenemos que ajustar las propiedades de nuestro proyecto a la plataforma y las características de nuestra aplicación.

En nuestro caso, tuvimos que introducir algunos cambios en estos aspectos, a medida que las aplicaciones iban creciendo y usando distintos recursos.

Empezamos desarrollando una aplicación que implementaba un AG clásico sobre la CPU. Para ello se procuró usar las clases más estándar posibles de C++, a fin de evitar problemas de incompatibilidades en el futuro.

Sin embargo, para conseguir lanzar dos algoritmos de este tipo en paralelo, necesitamos introducir la clase específica CWinThread, perteneciente a Microsoft Foundation Classes (MFC).

Por otra parte, se desarrolló el algoritmo que utiliza la GPU. En este caso tuvimos que incluir las dependencias específicas derivadas del uso de OpenGL. Para ello, se introdujeron las dependencias adicionales de estas librerías en las propiedades de entrada del vinculador.

El problema surgió cuando integramos los dos proyectos anteriores. Encontramos un error en el vinculador que nos advertía de símbolos definidos por duplicado[1]:

```
nafxcwd.lib(afxmem.obj) : error LNK2005: ya se definió "void * __cdecl
operator new(unsigned int)" (??2@YAPAXI@Z) en libcpmtd.lib(newop.obj)
```

Lo primero de lo que hay que darse cuenta es del tipo de biblioteca de tiempo de ejecución que usamos. En el caso de tener una aplicación sin hilos, las bibliotecas de proceso simple nos valen, cosa que ocurría en la implementación GPU. Sin embargo, el

[1] *Ayuda de Visual C++.NET: "Error en las herramientas del vinculador LNK2005"*

paralelismo que queríamos dar a la aplicación obligaba a cambiarlas por las bibliotecas de multiproceso. Es conveniente adecuar estas bibliotecas a las características de la aplicación que queremos desarrollar, puesto que si no podemos encontrar problemas de símbolos repetidos[1].

Otra de las formas de resolver este problema es eliminar del linkado toda librería superflua que no usemos y que, sin embargo, puede estar incluida en la generación por defecto. Para ello hay que excluir la librería causante del problema de las librerías por defecto que usa Visual para generar la aplicación[2].

También podría tomarse la precaución de asegurarse de que las clases de MFC se linkan antes que las estándar. Esto se debe a que MFC usa *vínculos fuertes* en el linkado de símbolos, mientras que las estándar usan *vínculos débiles*. Para ello podemos forzar al vinculador a hacerlo (opciones de proyecto) o incluir las cabeceras de las MFC en el archivo de cabecera precompilado[2].

En nuestro caso, el problema se derivaba de la inclusión de la cabecera <*iostream*> para las clases de la aplicación GPU (sin integrar). Nuestra biblioteca de clases en tiempo de ejecución (CRT) parecía incluir las nuevas clases estándar de C++. Dentro de ellas se incluye una versión de <*iostream*>. Al parecer, incluíamos también la versión antigua, con lo que los símbolos aparecían definidos varias veces[3,4].

A3.2 Problema de longitud de cromosoma para GPU

En un AG clásico sobre CPU, el valor de la longitud de cromosoma no tiene ninguna restricción específica. Sin embargo, esto cambia sobre la GPU según sea la estructura de datos considerada sobre la textura, y según la implementación de los operadores genéticos.

El cromosoma de un individuo está guardado en píxeles de la textura. En nuestro caso, cada gen ocupa una de las componentes del píxel. Por tanto, cada píxel tiene cuatro genes del cromosoma. Nuestro algoritmo funciona con longitudes de cromosoma múltiplos de 4.

Solución 1

Supongamos ahora que nuestro *lcrom* no es múltiplo de 4, por ejemplo elegimos 10. Para introducir las dos zonas de genes dentro la textura, meteríamos 20 genes, es decir, 5 píxeles. El problema es que no podemos dividir estos cinco píxeles en dos zonas de

---

[1] *Ayuda de Visual C++.NET: "PRB: LNK2005 Errors When Link C Run-Time Libraries Are Linked Before MFC Libraries"*

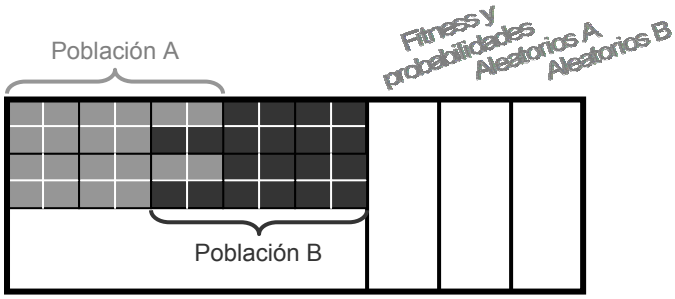
[2] *Ayuda de Visual C++.NET: "Error en las herramientas del vinculador LNK2005"*

[3] *Ayuda de Visual C++.NET: "INFO: What Are the C/C++ Libraries My Program Would Link With?"*

[4] *Ayuda de Visual C++.NET: "INFO: Standard C++ Library Frequently Asked Questions"*

genes bien definidas. Nuestro individuo de la población A tendría dos genes en dos componentes del tercer píxel, mientras que el de la población B tendría dos genes en dos componentes del mismo píxel. Así no desperdiciamos memoria de la GPU.

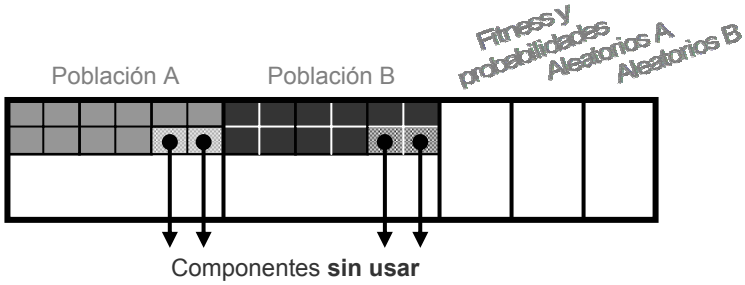
Para soportar este solapamiento de zonas tendríamos que implementar con mucho cuidado los *fragment programs*, introduciendo en ellos distinciones según en caso. Esto podría recortar la eficiencia de cómputo de la tarjeta.



Solución 2

Otra solución sería la de definir bien la división entre zonas. En este caso, necesitamos tres píxeles para cada individuo, donde entran diez genes, pero nos sobran dos componentes de uno de los píxeles. De esta forma, para cada fila de la zona de genes nos sobran cuatro componentes de píxel (dos por cada individuo de la fila de una zona, cuatro porque hay dos zonas), pero conseguimos una distinción clara entre las dos poblaciones.

El problema de esta aproximación es que desperdiciamos memoria de la tarjeta, un recurso muy valioso en este caso.



Los problemas ocasionados por ambas soluciones parecen mayores que la ventaja que se puede obtener. La restricción de multiplicidad de cuatro es muy fácil de cumplir y simplifica el algoritmo, de modo que optamos por mantenerla.

A3.3 Problema del tamaño de la Población

En principio, no habría ningún problema en usar el tamaño de población que quisiéramos para ambas poblaciones. De hecho, no es más recomendable un tamaño de población que otro, al contrario que en el caso de la longitud de cromosoma (problema descrito en el apartado anterior), para el que la opción más aconsejable es la de usar múltiplos de 4.

Sin embargo, nuestros algoritmos genéticos sólo funcionan con un tamaño de población par. Y esto es válido tanto para el de CPU como para el de GPU.

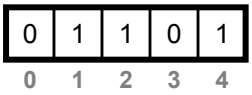
Esto se debe únicamente al tipo de implementación del operador genético de cruce. Como hemos explicado, el cruce se hace entre individuos consecutivos, de dos en dos. Por tanto, debe haber un número par de individuos para la población. Si no fuera así, el último individuo de la población no tendría con quién cruzarse, y es de esperar que se produzca una excepción en la aplicación.

A3.4 Visualización de los Genes del Individuo

Tanto en el algoritmo de CPU como en el de GPU, existen implementados procedimientos que muestran por pantalla los genes de cada uno de los individuos de la población. Estos procedimientos son muy útiles a la hora ver el estado de la población del algoritmo en un instante de ejecución concreto, y se han utilizado para comprobar el correcto funcionamiento de los diferentes operadores genéticos.

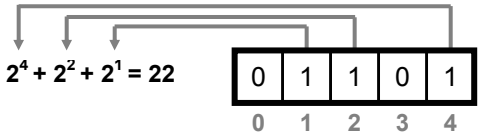
A continuación explicamos el modo en que se muestran los cromosomas, y que hay que tener en cuenta si se utilizan estos métodos auxiliares.

La visualización de los genes se hace comenzando por el gen de la posición cero a la izquierda, por lo que un individuo se visualiza del siguiente modo:



En el caso de estar trabajando con una función de evaluación matemática, debemos tener cuidado. Para estas funciones se trabaja con puntos del espacio codificados en binario. Por tanto, no es lo mismo 1011 que 1101.

Según nuestra visualización, un número comienza con el bit más significativo a la izquierda y el más significativo a la derecha:



A3.5 Coordenadas de textura (GPU)

En los *fragment programs* debemos pasar las coordenadas de las multitexturas, para conseguir los datos que necesitamos mediante una llamada a texRECT.

En algunos *fragments*, por ejemplo para el cruce, necesitamos los datos del individuo y los del individuo siguiente dentro de la población. Para cada píxel, entonces, necesitamos el píxel mapeado (píxel del individuo) y el píxel del siguiente individuo. Para poder acceder a él, considerada la textura como un array bidimensional, conseguiríamos su coordenada de textura sumando 1 a la columna de la coordenada del píxel mapeado.

Sin embargo, cuando llamamos a texRECT con la coordenada modificada del modo descrito anteriormente, el resultado obtenido no se corresponde con el esperado.

Si mostramos en la pantalla estas coordenadas, podemos observar que no suelen ser números enteros. Nosotros pensamos en la textura como un array bidimensional en las que las posiciones de los píxeles van del 0 hasta tamaño – 1. Sin embargo, parece que internamente, las coordenadas de las multitexturas pasadas al *fragment* toman desfases decimales, dependiendo del tamaño de las zonas mapeadas.

Creemos que se debe a que la tarjeta trabaja con coordenadas de ventana, y no con coordenadas de textura como estructura de datos. A su vez, esto se debe al *solapamiento*, un mecanismo con el que la GPU realiza el paralelismo de ejecución sobre píxeles.

Mecanismo de solapamiento

Cuando se definen multitexturas de lectura, éstas se mapean sobre la zona de escritura, a fin de poder acceder a los datos de entrada del *fragment program*. No se exige que las diferentes zonas de lectura tengan las mismas dimensiones que la zona de escritura. En caso de que no sean iguales, las zonas de lectura deben ser redimensionadas a la zona de escritura para poder dar soporte al paralelismo.

Cuando usamos tal cual las coordenadas recibidas como parámetros de entrada del *fragment program* en funciones Cg (como texRECT), no tenemos ningún problema, a pesar del desfase, y la textura es capaz de reconocer la posición indicada correctamente.

Sin embargo, hemos comprobado que cuando copiamos esta coordenada (en otra variable) y la modificamos, para poder acceder a otra posición relativa, texRECT no reconoce la posición, advirtiéndonos en la mayoría de los casos que está vacía.

Para solucionar este problema, tenemos que corregir el desfase introducido por el solapamiento.

Corrección del desfase de coordenadas

Una vez copiada la coordenada, podemos truncar su parte decimal. Conviene saber que el desfase es sólo decimal, es decir, que la parte entera es realmente la posición que queremos. Para ello podemos usar *modf*, para truncar la parte decimal, o hacerlo con

alguna otra función de Cg equivalente (*floor*). La nueva coordenada sí es reconocida correctamente por texRECT.

Otra forma de solucionar el problema es pasar un parámetro al *fragment program*, indicando cuál debe ser la corrección del desfase. Para esto, hay que ser cuidadoso y asegurarse de que las dimensiones de las multitexturas de lectura y la textura de escritura son iguales en cada llamada.

Ambas soluciones han sido implementadas en diferentes Cgs. Así por ejemplo, "fp_numalea" utiliza la corrección de desfase por parámetro y "fp_cruce" usa corrección mediante truncamiento con *modf*.

A3.6 Comparaciones de datos sobre GPU

Cuando estamos implementando un *fragment program* sobre Cg, debemos tener especial cuidado con los tipos que usamos.

A la hora de comparar números, tuvimos problemas porque parecía que el programa no era capaz de compararlos bien. Por ejemplo, cuando comparábamos un float con 0, no siempre se conseguía el resultado esperado, aun cuando el float valiese cero.

Esto se debe a que Cg no soporta la misma compatibilidad de tipos que C, de modo que habrá que prestar siempre especial atención a las comparaciones de datos.

Para comparar valores float con entero hemos recurrido a operadores de relación $<$ ó $\leq$, y a funciones de redondeo y truncamiento.

A3.7 Incompatibilidad de OpenGL con Hilos

En el diseño inicial de la ejecución de dos hilos en paralelo, pretendíamos lanzar un hilo con la ejecución del algoritmo de GPU. Sin embargo, como se ha explicado, tuvimos que modificar este diseño debido a los problemas de compatibilidad encontrados entre OpenGL e hilos.

El problema reside en el contexto de OpenGL. Cuando creábamos el algoritmo de GPU, lo hacíamos en el hilo principal de la aplicación. Éste algoritmo se encarga, en su construcción, de inicializar las librerías OpenGL, la ventana de visualización y el contexto OpenGL. Este contexto no es compartido, bajo Windows, por los distintos hilos creados.

Al crear el hilo y ejecutar el algoritmo de GPU en él, nos daba fallos en las llamadas a funciones de la librería OpenGL.

Cuando un contexto OpenGL es creado, se asigna al hilo que lo crea. Cuando intentábamos usar las funciones OpenGL sobre el hilo secundario (el creado), nos daban fallos. Esto se debe a que dichas funciones no pueden trabajar sin un contexto

inicializado. Efectivamente, ese contexto no existía para el hilo creado, por lo que se producían los errores en las llamadas.

Una posible solución sería hacer que los dos hilos que trabajan en la aplicación (el de la aplicación y el hilo creado) compartiesen el mismo contexto. Otra, que es la que implementamos, era ejecutar el AG de GPU sobre el hilo de la aplicación.

Otras soluciones posibles pasaban por modificar el diseño de las clases de algoritmos, y permitir que fuera el hilo quien crease el objeto GPU. El diseño de la clase que implementa el hilo secundario necesita una función de ejecución y un objeto que la contenga. Si creamos una clase que implemente una función, la cual se encargue de crear el AG de GPU, podemos inicializar el contexto OpenGL sobre el propio hilo secundario. Sin embargo, no nos parecía muy elegante usar una clase intermedia vacía, que sólo ejecutase la función de crear el hilo.

Bibliografía

[1] Whitley, L. D.: Fundamental principles of deception in genetic search. In: Rawlins, G., (ed.): Foundations of Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, (1991) 221-241

[2] Goldberg, D.: Genetic Algorithms and Walsh Functions: Part II, Deception and its Analysis. Complex Systems 3 (1989) 153-171

[3] Holland, J.H.: Adaptation in Natural and Artificial Systems. Ann Arbor, MI: University of Michigan Press (1975).

[4] Mathias K., Whitley, L.D.: Transforming the search space with gray coding. In Proc. IEEE Int'l Conference on Evolutionary Computation (1994) 513-518

[5] Yokose, Y., Cingoski, V., Kaneda K., and Yamashita, H.: Performance Comparison Between Gray Coded and Binary Coded Genetic Algorithms for Inverse Shape Optimization of Magnetic Devices, Applied Electromagnetics, (2000) 115-120

[6] Wolfram, Gray Code, http://mathworld.wolfram.com/GrayCode.html

[7] Goldberg, D.: Genetic Algorithms and Walsh Functions: Part I, A Gentle Introduction. Complex Systems 3 (1989) 129-152

[8] Goldberg, D., Bridges, C. An Analysis of a Reordering Operator on a GA-Hard Problem. Biological Cybernetics 62 (1990) 397-405

[9] Dasgupta, D.: Handling deceptive problems using a different genetic search. In Proceedings of the First IEEE Conference on Evolutionary Computation (1994) 807-811

[10] Shackleton M., Shipman, R., and Ebner, M.: An investigation of redundant genotypephenotype mappings and their role in evolutionary search. In Proceedings of Congress on Evolutionary Computation (2000) 493-500

[11] Whitley, D., Rana, S., Heckendorn, R.B.: Exploiting Separability in Search: The Island Model Genetic Algorithm. Journal of Computing and Information Technology, v. 7, n. 1, (1999) 33-47 (Special Issue on Evolutionary Computing)

[12] Sehitoglu, O.T., Ucoluk, G.: A building block favoring reordering method for gene positions in genetic algorithms. In Spector, L., Goodman, E.D., Wu, A. et.al., (eds.), Proceedings of the Genetic and Evolutionary Computation Conference (2001) 571-575

[13] De Jong, K.A.: An Analysis of the Behavior of a Class of Genetic Adaptive Systems, Ph.D. Thesis, University of Michigan (1975)

[14] Digalakis J., Konstantinos, M.: An Experimental study of Benchmarking Functions for Evolutionary Algorithms. International Journal of Computer Mathemathics, Vol. 79, (2002) 403-416

[15] Salomon. R.: Reevaluating Genetic Algorithm Performance under Coordinate Rotation of Benchmark Functions. BioSystems vol. 39, Elsevier Science (1995) 263-278

[16] Collard P., Aurand J.-P.: DGA: an efficient Genetic Algorithm. In Proceedings of the 11th European Conference on Artificial Intelligence (ECAI'94) (1994) 487-492

[17] Collard P., Clergue M., Defoin P.M.: Artificial Evolution: In Proceedings of the Fourth European Conference (AE'99), Lecture Notes in Computer Sciences 1829, Springer-Verlag Ed. (2000) 254-265

[18] Goldberg, D.E.: Simple genetic algorithms and the minimal, deceptive problem. In Davis, L., ed.: Genetic Algorithms and Simulated Annealing. Morgan Kaufmann, Los Altos, CA (1987) 7488

[19] Chow, Rick: Evolving Genotype to Phenotype Mappings with a Multiple-Chromosome Genetic Algorithm. K. Deb et al. (Eds): GECCO 2004, LNCS 3102, 1006-1017

[20] Fernando G. Lobo: A philosophical essay on life and its connections with genetic algorithms. (2001)

[21] Michalewicz, Zbigniew: Genetic Algorithms + Data Structures = Evolution Programs. Tercera edición. Ed. Springer. 1999.

[22] Hidalgo J.I., Lanchares J., Ibarra A.: Introducción a los Algoritmos Evolutivos. Versión 1.2. UCM. (2001)

[23] Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial Intelligence Through Simulated Evolution, John Wiley, (1966)

[24] Glover, F.: Heuristics for Integer Programming Using Surrogate Constraints, Decision Sciences, Vol. 8. (1977) 156-166

[25] Holland, J.H.: Adaptation in Natural and Artificial Systems, University of Michigan Press (1975)

[26] De Jong, K.A.: Genetic Algorithms: A 10 Year Perspective, Proceedings of the First International Conference on Genetic Algorithms, Lawrence Erlbaum Associates (1985) 169-177

[27] Davis, L. (Editor): Genetic Algorithms and Simulated Annealing, Morgan Kaufmann Publishers (1987)

[28] Grefenstette, J.J (Editor): Proceedings of the First International Conference on Genetic Algorithms, Lawrence Erlbaum Associates (1985)

[29] Grefenstette, J.J (Editor): Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates (1987)

[30] Groves, L., Michalewicz, Z., Elia, P., Janikow, C.: Genetic Algorithms for Drawing Directed Graphs, Proceedings of the Fifth International Symposium on Methodologies of Intelligent Systems, Amsterdam (1990) 268-276

[31] Michalewicz, Z., Vignaux, G.A., Hobbs, M.: A Non-Standard Genetic Algorithm for the Nonlinear Transportation Problem, ORSA Journal on Computing, Vol. 3 (1991) 307-316

[32] Smith, S.F.: A Learning System Based on Genetic Algorithms, PhD Dissertation, University of Pittsburgh (1980)

[33] Smith, S.F.: Flexible Learning of Problem Solving Heuristics through Adaptive Search, Proceedings of the Eighth International Conference on Artificial Intelligence, Morgan Kaufmann Publishers (1983)

[34] Vignaux, G.A., Michalewicz, Z., A Genetic Algorithm for the Linear Transportation Problem, IEEE Transactions on Systems, Man and Cybernetics, Vol. 21 (1991) 445-452

[35] Koza, J.R.: Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems, Standford University (1990)

[36] De Jong, K.A.: On Using Genetic Algorithms to Search Program Spaces, Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates (1987) 210-216

[37] De Jong, K.A.: Learning with Genetic Algorithm: An Overview, Machine Learning, Vol. 3 (1988) 121-138

[38] Davis, L., Steenstrup, M.: Genetic Algorithms and Simulated Annealing: An Overview, Genetic Algorithms and Simulated Annealing, Morgan Kaufmann Publishers (1987) 1-11

[39] Antonisse, H.J., Keller, K.S.: Genetic Operators for High Level Knowledge Representation, Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates (1987) 69-76

[40] Forrest, S.: Implementing Semantic Networks Structures Using the Classifier System, Proceedings of the First International Conference on Genetic Algorithms, Lawrence Erlbaum Associates (1985) 24-44

[41] Fox, B.R., McMahon, M.B.: Genetic Operators for Sequencing Problems, First Workshop on the Foundations of Genetic Algorithms and Classifier Systems, Morgan Kaufmann Publishers (1991) 284-300

[42] Grefenstette, J.J.: Incorporating Problem Specific Knowledge into Genetic Algorithms, Genetic Algorithms and Simulated Annealing, Morgan Kaufmann Publishers (1987) 42-60

[43] Starkweather, T., McDaniel, S., Mathias, K., Whitley, C., Whitley, D.: A Comparison of Genetic Sequencing Operators, Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers (1991) 69-76

[44] Davis, L. (Editor): Handbook of Genetic Algorithms, Van Nostrand Reinhold (1991)

[45] Brooker, L.B.: Improving Search in Genetic Algorithms, Genetic Algorithms and Simulated Annealing, Morgan Kaufmann Publishers (1987) 61-73

[46] Bethke, A.D.: Genetic Algorithms as Function Optimizers, Doctoral Dissertation, University of Michigan (1980)

[47] Bertoni, A., Dorigo, M.: Implicit Parallelism in Genetic Algorithms, Artificial Intelligence, Vol. 61 (1993) 307-314

[48] Bäck, T., Hoffmeister, F., Schwefel, H.: A Survey of Evolution Strategies, Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers (1991) 2-9

[49] Holland, J.: Adaptation in Natural and Artificial Systems, University of Mitchigan Press (1975)

[50] Tanese, R.: Distributed genetic algorithms, Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers (1989) 434-440

[51] Goldberg, D.E.: Genetic Algorithms in search, optimization and machine learning, Addison Wesley (1989)

[52] Tanese, R.: Parallel Genetic Algorithms for a hypercube, Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates (1987) 177-183

[53] Coello, C.A., Christiansen, A.D.: Moses: A multiobjetive optimization tool for engineering design, Engineering Optimization, 31 (1999) 337-368

[54] Coello, C.A.: A Comprehensive survey of evolutionary-based multiobjetive Optimization Techniques, Knowledge and Information Systems, 1 (1999) 269-308

[55] Parmee, I.C., Cvetkovic, D., Watson, A.H., Bonham, C.R.: Multiobjetive satisfaction within an interactive evolutionary design environment, Evolutionary Computation, 8 (2000) 197-222

[56] Rudolph, G.: Convergence analysis of canonical Genetic Algorithms, IEEE Transactions on Neural Networks, 5 (1994) 96-101

[57] Aguirre, H., Tonaka, K., Sugimura, T.: Cooperative crossover and mutation operators in Genetic Algorithms, Proceedings of the 1999 Genetic and Evolutionary Computation Conference, Morgan Kaufmann (1999) 772

[58] Aguirre, H., Tonaka, K., Sugimura, T., Oshita, S.: Cooperative competitive model for genetic operators: Contributions of extinctive selection and parallel genetic operators, Proceedings of the 2000 Genetic and Evolutionary Computation Conference Late Breaking Papers, (2000) 6-14

[59] Hidalgo, J.I., Lanchares, J., Hermida, R.: Partitioning and placement for multi-fpga systems using Genetic Algorithms, Proceedings of the Euromicro DSD 2000

[60] Goldberg, D.E., Deb, K., Korb, B.: Don´t worry, be messy, Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers (1991) 24-30

[61] Rawlins, G.: Foundations of Genetic Algorithms, Morgan Kaufmann (1991)

[62] Vose, M.D., Liepins, G.E.: Schema disruption, Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers (1991)

[63] Whitley, L.D.: Foundations of Genetic Algorithms 2, Morgan Kaufmann (1993)

[64] Whitley, L.D., Vose, M.D.: Foundations of Genetic Algorithms 3, Morgan Kaufmann (1995)

[65] Vose, M.D.: The Simple Genetic Algorithm. Foundations and Theory, MIT Press (1999)

[66] Vose, M.D.: Modeling Simple Genetic Algorithms, Evolutionary Computation, Vol. 3 (1995) 453-472

[67] Juliany, J., Vose, M.D.: The Genetic Algorithm fractal, Evolutionary Computation, Vol. 2 (1994) 165-180

[68] Vose, M.D., Wright, A.H.: Simple Genetic Algorithms with linear fitness, Evolutionary Computation, Vol. 2 (1994) 347-368

[69] Liening, J.: A parallel genetic algorithm for performance-driven VLSI routing, IEEE Transactions on Evolutionary Computation, Vol. 1 (1997) 29-39

[70] Muhlenbein, H.: Parallel Genetic Algorithms, population genetic and combinatorial optimization, Parallel Problem Solving from Nature – Proceedings of the First Workshop PPSN, Vol. 496 (1991)

[71] Prieto, M.: Paralelización de Métodos Multimalla Robustos, PhD thesis, UCM (2000)

[72] High performance fortran, http://www.crpc.rice.edu/HPFF/

[73] The openmp application program interface, http://www.openmp.org

[74] Mpi: A message passing interface standard, http://www.mpi-forum.org

[75] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.S.: PVM: Parallel Virtual Machine. A users guide and tutorial for network parallel Computers, MIT Press Book (1994)

[76] Dorigo, M., Maniezzo, V.: Parallel Genetic Algorithms: Introduction and overview of current research, IOS Press (1993) 5-42

[77] Bianchini, R., Brown, C.M.: Parallel Genetic Algorithm on distributed-memory Architectures, Technical Report TR 436, Computer Sciences Department University of Rochester (1993)

[78] Manderick, B., Spiessens, P.: Fine-grained parallel genetic algorithms, Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers (1989) 428-433

[79] Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms, Kluwer Academic Publishers (2000)

[80] Bianchini, R., Brown, C.M.: Parallel Genetic Algorithm on distributed-memory Architectures, Transputer Research and Applications, Vol. 6, IOS Press (1993) 67-82

[81] Cantu-Paz, E.: A summary of research on parallel genetic algorithms, Technical Report 95007, University of Illinois (1995) http://gal4.ge.uiuc.edu/illigal.home.html

[82] Cohoon, S., Hedge, J., Martin, S., Richards, D.: Punctuated equilibria: a parallel genetic algorithm, IEEE Transaction on CAD, Vol. 10 (1991) 483-491

[83] Cg Toolkit: User's Manual. A Developer's Guide to Programmable Graphics. NVIDIA Corporation. (2002) ftp://download.nvidia.com/developer/cg/Cg_Users_Manual.pdf

[84] Richard J. Povinelli: Comparing Genetic Algorithms Computacional Performance Improvement Technique

[85] K.C. Tan, M.L. Wang, W. Peng: A p2p genetic algorithm environment for the internet, ACM Press (2005)

[86] Especificación técnica de GPU nvidia GeForce FX 5950 Ultra: http://www.nvidia.com/page/fx_5700.html

[87] AOpen Inc.: Tecnología "Hyper-Threading" (2004)

[88] Exploring Hyper-Threading Performance - Technical Details, publicado en 2CPU.com, (2004).

[89] XML Tutorial, FunctionX,Inc (2004), http://www.functionx.com/vcnet/xml/Lesson01.htm hasta http://www.functionx.com/vcnet/xml/Lesson09.htm

[90] Visual C++ _NET XML Reading and Writing, FunctionX,Inc (2004), http://www.functionx.com/vcnet/xml/readwrite.htm

[91] Wong, D.F., Leong, H.W., Lin, C.L.: Simulated Annealing for VLSI Design, Klumer

[92] A E.-G. Talbi, A P. Bessi, A parallel genetic algorithm for the graph partitioning problem, Proceedings of the 5th international conference on Supercomputing, Cologne, West Germany, ACM Press (1991), 312-320

[93] J.I. Hidalgo, M. Prieto, J. Lanchares, F.Tirado: A Parallel Genetic Algorithm for solving the Partitioning Problem in Multi-FPGA systems. Proceedings of 3rd international Meeting on vector and parallel processing. Oporto (Portugal), 21-23 de Junio de 1998, 717-722

[94] Herrera, E. Huedo, R.S. Montero, and I.M. Llorente: A Grid-Oriented Genetic Algorithm, in P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, Springer-Verlag, (2005) 315–322